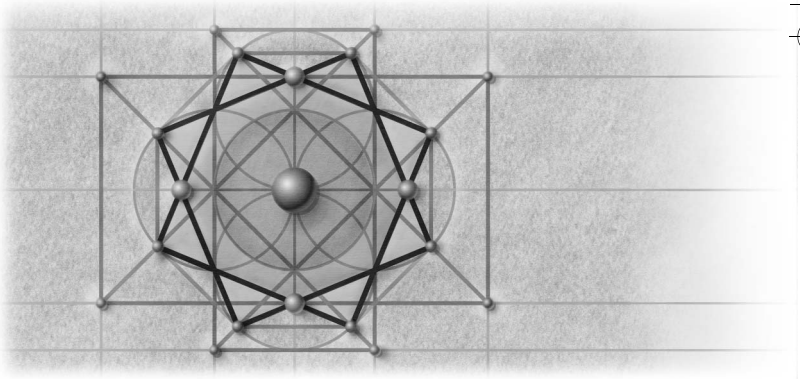

8

Errors, Warnings, and Issues

This chapter looks at the different errors, warnings, and issues the Upgrade Wizard generates when you upgrade your Visual Basic 6 projects to Visual Basic .NET.

Each time you upgrade a project, the wizard upgrades most of the code and objects to Visual Basic .NET, but some items are not automatically changed. These items require manual modifications after the wizard finishes. To understand why, consider this example. In Visual Basic .NET, the *Caption* property of a label is now called the *Text* property. Label.Caption maps to Label.Text. The Upgrade Wizard knows about this and changes all *Label.Caption* references to *Label.Text*. But what happens if the wizard can't determine whether a particular object is indeed a label? When the wizard isn't able to determine what data type an object is, it inserts a warning into your code. The following code is a good example. Assume *Form1* is a form, and *Label1* is a Label control:

```
Dim o As Variant
Set o = Form1.Label1
o.Caption = "VB Rocks"
```

In this example, the variable *o* is declared as a variant and then assigned to *Label1*. The *Caption* property is then set to the string "VB Rocks." Because *o* is a variant and is late-bound, Visual Basic 6 doesn't know what type of object it is at design time. (This is why you don't get IntelliSense for variants at design

time.) Like the Visual Basic 6 Code Editor, the Upgrade Wizard also has a design-time view of the code. We'll follow what the wizard does as it walks through the three lines of code in our simple example:

```
Dim o As Variant
```

The first line is easy; the wizard knows to change *Variant* data types to *Object*, so it upgrades the first line as follows:

```
Dim o As Object
```

The second line is also upgraded automatically:

```
Set o = Form1.Label1
```

This is a *Set* statement, so it has to be an object assignment rather than a default property assignment. The variable *o* is assigned to the control Form1.Label1 and upgraded as follows:

```
o = Form1.DefInstance.Label1
```

The first two lines were upgraded automatically. The final line, however, causes problems:

```
o.Caption = "VB Rocks"
```

Because *o* is late-bound, the Upgrade Wizard doesn't know what to do with the *Caption* property. Perhaps *o* is a label, but perhaps it is another type of control. When the Upgrade Wizard finds code that can't be resolved at design time, it inserts a warning. The final line upgrades to the following:

```
'UPGRADE_WARNING: Couldn't resolve default property of object o.Cap-
tion. 'Click for more: ms-help://MS.MSDNVS/vbcon/html/vbup1037.htm

o.Caption = "VB Rocks"
```

What do you do now? If you run the code, it causes a run-time exception. Often, the best solution is to change the variable declaration to be strongly typed. In this example, change the first line to

```
Dim o As Label
```

After this change, the compiler will report that *Caption* is not a property of *Label* by generating a Task List entry and by highlighting the code with a green underline. You should then change the property to *Text*. Another good solution that avoids the problem entirely is to declare *o* as a label in Visual Basic 6.

Declaring it as a label makes it strongly typed, and the wizard will have no trouble changing *Caption* to *Text*. Here is how to rewrite the code in Visual Basic 6 to be strongly typed:

```
Dim o As Label
Set o = Form1.Label1
o.Caption = "VB Rocks"
```

After upgrading, the code becomes

```
Dim o As System.Windows.Forms.Label
o = Form1.DefInstance.Label1
o.Text = "VB Rocks"
```

This code works perfectly in Visual Basic .NET.

This is a simple example; we can figure out from looking at the code that *o* is a label. However, the wizard doesn't know this. It has a set of rules for converting Visual Basic 6 to Visual Basic .NET that it follows for each property, method, and event of a particular object type. When a variable is late-bound, the wizard doesn't know how to interpret its properties.

Couldn't the Upgrade Wizard simply guess the object type? Other examples show the futility of this strategy. In the following example, even the original author of the code wouldn't be able to determine what type of object *o* will be set to at run time. Let's assume that *Label1* is a label and *Text1* is a text box:

```
Dim o As Object
If someVariable = 1 Then
   Set o = Me.Text1
Else
   Set o = Me.Label1
End If
```

It's impossible to know whether *o* will be a label or a text box until the code actually executes.

EWI is the common term for the errors, warnings, and issues the Upgrade Wizard inserts into your code. EWIs alert you to problems like the preceding one. EWI comments are inserted on the line before the problem. The comment text is in two parts: the message and the link to Help. In the following code, the underlined part of the text is actually a hyperlink to the appropriate Help topic:

```
'UPGRADE_WARNING: Couldn't resolve default property of object o.Cap-
tion. 'Click for more: ms-help://MS.MSDNVS/vbcon/html/vbup1037.htm
```

All EWIs are associated with Help topics that show you how to fix the problem. Because the EWI is inserted as a comment, it doesn't affect the compilation or execution of your project.

Identifying All the Problems

Does the Upgrade Wizard detect every problem during an upgrade? Is every problem marked in your code with easy-to-follow guidance? Unfortunately, it's not quite as simple as that. Some problems might have only a 0.01 percent chance of occurring, so what should the wizard do—mark every occurrence of the code, generating false alarms in 99.99 percent of the cases, or ignore a potential problem and let it generate a compile or run-time error? The Upgrade Wizard walks a fine line in determining what it should treat as an error and what it shouldn't. It warns about most problems it finds, but as you'll see later in this chapter, there are some that it doesn't warn about.

The Different Kinds of EWIs

There are six types of EWIs. Let's look at the most serious type first.

Upgrade Issues

Upgrade issues are inserted into your code whenever the wizard meets code that will cause a compile error. Upgrade issues mark items that you need to fix before the program will run. A good example is the *OLEDrag* method. This method has no direct equivalent in Windows Forms, so the following code

```
Text1.OLEDrag
```

upgrades to

```
'UPGRADE_ISSUE: TextBox method Text1.OLEDrag was not upgraded
Text1.OLEDrag
```

Because the *OLEDrag* method is not a member of the *TextBox* object, the *Text1.OLEDrag* statement causes a compile error. The compile error shows in the Task List. The upgrade issue does not. Why not? If it did, you would have two Task List entries for every compiler error instead of one. As you've learned, at the end of every EWI comment is a link to a Help topic describing how to fix the problem. (In this case, you need to reimplement drag-and-drop capability using the new Windows Forms objects.) We've removed the hyperlinks from most of the other examples in this book to save space.

Upgrade ToDos

Upgrade ToDos let you know when code has been partially upgraded and needs finishing before it will run. This type of issue commonly arises when you declare a variable of a type that contains a fixed-size array. For example, the following code defines a type called *myType* that contains a fixed-size array. It then creates an instance of *myType* called *myVariable*.

```
Type myType
   myArray(10) As String
End Type
Sub Main()
   Dim myVariable As myType
End Sub
```

In Visual Basic 6, this code works without a problem. In Visual Basic .NET, the *Type* keyword is renamed *Structure*; arrays defined with the *Structure* keyword aren't initialized—you have to initialize them yourself. When upgrading this example, the wizard does what it can: It changes the *Type* keyword to *Structure* and upgrades the variables within the structure to Visual Basic .NET. It creates a *Sub* to initialize the arrays. In Visual Basic .NET, structures can now have methods, so the Upgrade Wizard also creates an *Initialize* method that initializes the arrays for you. The hard work is done. The only step remaining is to write code that calls the *Initialize* method. This is where you come in. The Upgrade Wizard inserts a ToDo EWI that tells you what you need to do. The upgraded code looks like this:

```
Structure myType
   Dim myArray() As String
   'UPGRADE_TODO: "Initialize" must be called to initialize instances
   'of this structure.
   Public Sub Initialize()
      ReDim myArray(10)
   End Sub
End Structure
Public Sub Main()
   Dim myVariable As myType
End Sub
```

To make the code work, you need to add a line in *Sub Main* to call the *Initialize* method. The following code shows the modified *Sub Main*:

```
Public Sub Main()
   Dim myVariable As myType
   myVariable.Initialize()
End Sub
```

After you make the modification, the code runs as it did in Visual Basic 6. Like upgrade issues, code marked with ToDo comments must be fixed before the program will run.

Run-Time Warnings

Run-time warnings alert you to behavior differences between Visual Basic 6 and Visual Basic .NET. A behavior difference is code that doesn't cause a compile error but might act differently at run time. For example, the *Dir* function is used to return the list of files or directories inside a particular directory. In previous versions of Visual Basic, *Dir* also returned . and .. (indicating current directory and parent directory). In Visual Basic .NET, the *Dir* function doesn't return the . and .. directories. When the following code is upgraded:

```
Dim myFilename As String
myFilename = Dir("C:\Temp\*.*", vbDirectory)
```

the *Dir* function is marked with an upgrade warning:

```
Dim myFilename As String
'UPGRADE_WARNING: Dir has a new behavior.
myFilename = Dir("C:\Temp\*.*", FileAttribute.Directory)
```

Some programs will work perfectly; some will have problems. If you have code that relies on *Dir* returning . and .., you need to make modifications. As with other EWIs, a link to Visual Basic .NET help explains the difference and tells you what to do next. Upgrade warnings show up in the Task List.

Design Issues

Design issues identify differences in the design of a form. For example, suppose you have a project with a form, and you've set the form's *OLEDropMode* property to *Manual*. When you upgrade the project, the *OLEDropMode* property is not upgraded, since there is no direct equivalent in Windows Forms. What happens to *OLEDropMode*? It is simply ignored and doesn't appear in the upgraded form. Because the code has no appropriate place to put this EWI, the Upgrade Wizard puts the following entry in the Form1 section of the upgrade report: "Form property Form1.OLEDropMode was not upgraded." The EWI is associated with a Help topic that explains how to implement drag-and-drop capability in Windows Forms. Because design issues are recorded only in the upgrade report, they do not appear in the Task List.

Upgrade Notes and Global Warnings

We've just looked at the four most common types of EWI. There are two other less common types of EWI, neither of which shows up in the Task List.

- Upgrade notes are inserted as friendly memos whenever code is substantially changed.
- Global warnings are inserted into the upgrade report to alert you to major issues, such as differences in data binding.

Understanding the Upgrade Report

The upgrade report summarizes any problems found during the upgrade. It is added to every project during an upgrade, whether or not there were any problems. Suppose you upgrade a project containing a form with a text box that implements drag and drop, and a module with a type that has a fixed-size array in it. The upgrade report might look something like Figure 8-1.

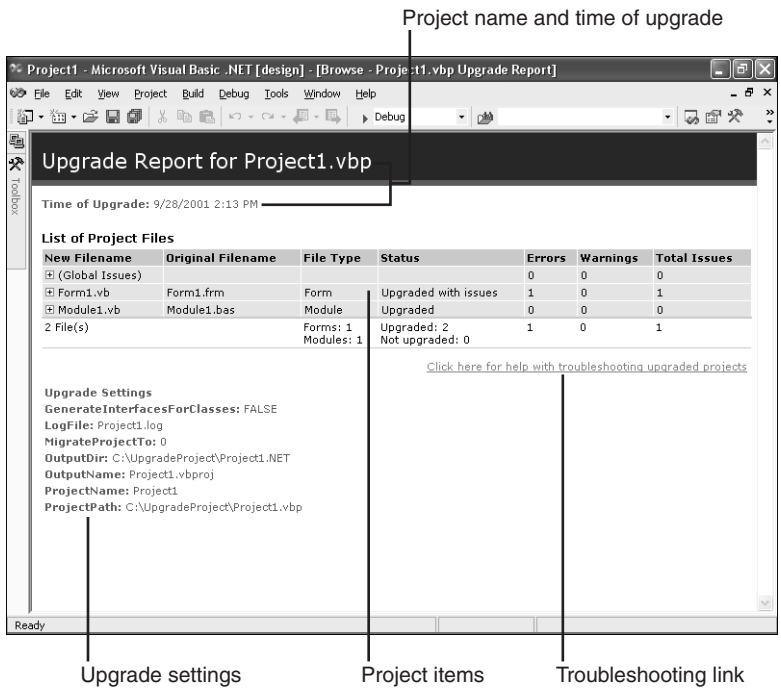

Figure 8-1 An upgrade report.

Let's look at the different parts of this report:

- **Project name and time of upgrade** The first line of the report gives the project name; the second line shows the date and time the project was upgraded.
- **Project items** The body of the report contains a table summarizing the project items (forms, classes, modules, and designers) and the issues found for each item during the upgrade. This table has seven columns.

  The first and second columns show the upgraded filename and the original filename, respectively. The third column gives the type of project item—form, class, module, and so on. The fourth column summarizes what happened during upgrade: Upgraded (no issues found), Upgraded with Issues, or Not Upgraded. If the project item was not upgraded, the original Visual Basic 6 file will simply be added to the project as a related document. The fifth and sixth columns show the count of errors and warnings for each project item, and the seventh column adds these two numbers to give the total number of issues.

  The table has a row for each project item, plus an extra row at the top of the table labeled (Global Issues) that notifies you of any project-level issues (such as general differences in data binding). Each row has a plus symbol on the left. If you click the plus symbol beside Form1, you'll see the individual issues, as shown in Figure 8-2.

List of Project Files

New Filename	Original Filename	File Type	Status	Errors	Warnings	Total Issues
⊞ (Global Issues)				0	0	0
⊟ Form1.vb	Form1.frm	Form	Upgraded with issues	1	0	1

Upgrade Issues for Form1.frm:

#	Severity	Location	Object Type	Object Name	Property	Description
1	Design Error	(Layout)	VB.TextBox	Text1	OLEDragMode	VB.TextBox property Text1.OLEDragMode was not upgraded.

⊞ Module1.vb	Module1.bas	Module	Upgraded	0	0	0
2 File(s)		Forms: 1 Modules: 1	Upgraded: 2 Not upgraded: 0	1	0	1

Figure 8-2 Expanded view of Form1 issues.

Form1 has one design error. The *OLEDragMode* property of TextBox1 was not upgraded. In the report, the description of each error has a second purpose. The description is also a hyperlink that navigates to Help for the EWI, just as the EWI comments in code do.

- **Troubleshooting link** This hyperlink navigates to general Help on troubleshooting upgrading. The Help topic also links to pages on the Internet offering the latest troubleshooting information.
- **Upgrade settings** The final section of the report shows the Upgrade Wizard settings, locations of the Visual Basic 6 and Visual Basic .NET projects, and any options that were set.

Estimating Fix Time

How long will it take to get your upgraded project working in Visual Basic .NET? Unfortunately, this chapter cannot give you a guaranteed length of time that a given project will take. What we can give you is a mechanism for loading the list of issues into Microsoft Excel and the reassurance that after doing a few upgrades, you'll be faster at solving issues and more accurate at knowing which issues are quick to solve and which take time.

The upgrade report is formatted to be easy to read. For large projects, in which you want to manage the distribution and status of issues within a group of developers, loading the issues into a spreadsheet might make more sense. Doing so allows you to add comments against each issue, such as who the issue is assigned to and its status. Let's look at how to do this.

The upgrade report is actually based on an underlying XML log file. The Upgrade Wizard creates a log file for every upgraded project. The log—called <projectname>.log—is placed in the same directory as the new Visual Basic .NET project. This log contains all the information in the upgrade report. In fact, the upgrade report is actually generated from the log file. The XML log file is easy for software to read but awkward for humans. Using a simple Excel macro, we can load the list of issues into a spreadsheet. Follow these steps:

1. Start Excel, and create a new workbook.
2. Add a new CommandButton to the worksheet. To do this, select View, Toolbars, Control Toolbox. Draw a CommandButton onto the spreadsheet.
3. Right-click the CommandButton, and choose Properties from the context menu.

4. In the Properties dialog box, change the name of the CommandButton to LoadUpgradeReport. Change the caption to read **Load Upgrade Report**.

5. Double-click the button to open the VBA Macro Editor. Enter the LoadUpgradeReport code.

6. Choose References from the Tools menu. In the References dialog box, select Microsoft XML, v3.0, and click OK.

7. Return to the worksheet. Click the Exit Design Mode button, and then click the Load Upgrade Report button. The macro prompts you for the filename of the log file. Enter the path to the log file. The macro populates the worksheet with the contents of the log file, as shown in Figure 8-3.

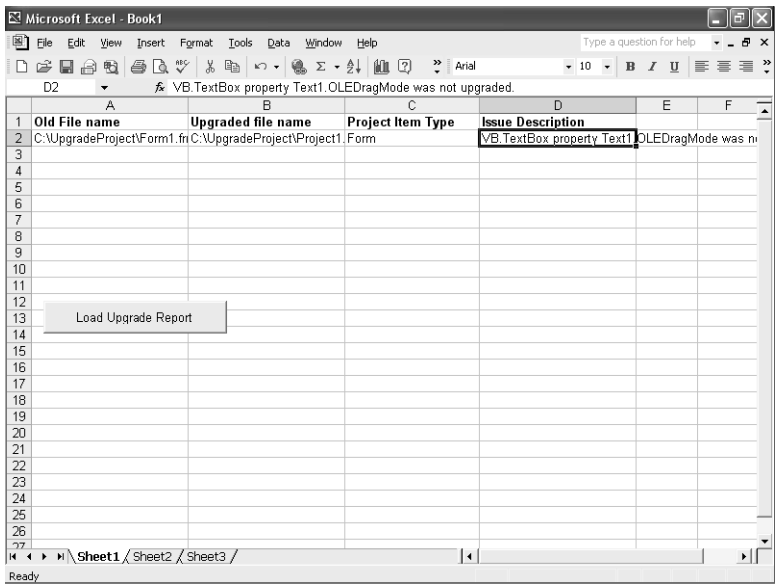

Figure 8-3 Loading the upgrade log into Excel.

Comments in the Excel code also show other columns that you can add to the spreadsheet, such as object type, object name, and the underlying EWI number.

Working with EWIs

Now that we've discussed the types of EWI that will be generated during upgrade, let's look at how to use EWIs to get your project working. At this point, you might have a few questions: How many issues should I expect to see in my upgraded project? How do I know when I've made the right fix? How do I keep track of what is left to fix?

The number of issues depends entirely on your programming style and the size of the project items. If you've followed all the recommendations in Chapter 4 you might minimize the number of EWIs to fewer than 10, or perhaps 0, per file. As a rule of thumb, forms usually have more issues than classes and modules, since forms have a rich object model and must be changed significantly when upgraded to Windows Forms.

The best idea is to get your project running as quickly as possible, leaving the run-time differences, noncritical work, and improvements until later. Fix the compile errors and ToDo items first. That way you'll be working with a project that is "runnable" as quickly as possible. You can filter the Task List to show all the EWIs and compile errors or to show only the compile errors. Start by filtering the Task List to show only compile errors. For information on filtering the Task List, see Chapter 6

A good method for working with EWIs is to fix a problem and then remove the EWI in the code so it stops showing up in the Task List. This ensures that the Task List shows only the issues left to resolve. Some compile errors can also be postponed. Your application might have code that performs a noncritical function, such as drawing a picture onto a PictureBox. You could decide to comment this out and add a remark such as the following:

```
'TODO: Fix Paint Code later
```

These ToDo comments will show up in the Task List when you change the filter to show all tasks. You can take care of noncritical functionality later, after you've done the important work. The main advantage of getting the project into a runnable state as soon as possible is psychological: it's a satisfying feeling knowing that your project is basically working, even if you have a few modifications left to make.

How do you know which modification you need to make for each EWI? The Visual Basic .NET Help will assist you with this. In the beginning, the modifications might take quite some time to fix, since you're learning about upgrading and Visual Basic .NET at the same time. The good news is that after you've fixed a problem the first time, you'll find that fixing the same problem again is much quicker, since you know what you're doing.

Performing a Two-Pass Upgrade

Open an old Visual Basic 6 project. Which EWIs do you think would be generated if you upgraded it to Visual Basic .NET? If only you had a way to know before upgrading! Perhaps you would change late-binding code or other easy-to-fix problems before upgrading. In these situations a two-pass upgrade is useful. The concept of a two-pass upgrade is simple: upgrade the project twice. The purpose of the first upgrade is simply to generate an upgrade report to identify where the problems are. You discard the first upgraded project after examining the report. Using the upgrade report, you fix some of the issues in Visual Basic 6 before upgrading a second time. The second pass is the real upgrade. When you do the second upgrade, you replace the first upgraded project, and you should have fewer issues to fix than in the first pass. In many cases, the two-pass approach significantly speeds upgrading.

The Different Upgrade EWIs

The Upgrade Wizard can generate 50 different EWIs. Let's examine each one, grouped by EWI type.

Upgrade Issues

- ***<statement> <statementname>* is not supported** Marks the *Calendar* statement, *Load <formname>* statement, *GoSub*, or *On <expr> GoSub*. The workaround for *Calendar* is to create a class that inherits from the *System.Globalization.Calendar* class. You can replace *Load <formname>* with code that creates a new instance of the form. *GoSub* is best fixed by reimplementing the routines as procedures.
- ***<functionname>* function is not supported** Added when the following functions are used: *CVErr*; *IMEStatus*; *AscB*, *MidB*, *ChrB,* and the other B functions; *VarPtr*, *ObjPtr,* and *StrPtr*.
- ***DoEvents* does not return a value** Added when code uses the return value of *DoEvents*. (In Visual Basic 6 it returns the number of open forms.) Although .NET provides no way to get the number of

open forms, you can implement your own forms collection to do this. See Chapter 15 for more details.

- **Declaring a parameter *As Any* is not supported** Added when an API has a parameter with a type of *As Any*. See Chapter 11for tips on fixing API problems.
- **Declaration type not supported: <*declaration type*>** Added to variable declarations not supported in Visual Basic .NET: arrays of fixed-length strings, references to a private enumerator from within a public enumerator, enumerator values assigned to color constants, arrays with more than 32 dimensions, arrays with negative upper bounds, and *Font* objects declared *WithEvents*.
- **Constant <*constantname*> was not upgraded** Some constants can't be upgraded to Visual Basic .NET.
- **Unable to determine which constant to upgrade <*constantname*> to** Added when a constant could upgrade to one of several Visual Basic .NET constants and the wizard can't determine from the context which constant is appropriate.
- ***LSet* cannot assign one type to another** Added when code uses *LSet* to assign a user-defined type from one type to another. The solution is to add a custom *CopyTo* method to the structure that copies the property in the source structure to corresponding properties in the destination structure.
- **COM expression not supported: Module methods of COM objects** Added when code uses a module method. This issue is discussed in Chapter 13
- **Objects of type *vbDataObject* are not supported** Added to the *VarType* function when the Upgrade Wizard detects that you're testing for *VbVarType.vbDataObject* (value 13). Objects that don't support the *IDispatch* interface can't be used in Visual Basic .NET. This is a rare issue.
- **<*objecttype*> object <*object*> was not upgraded** Added to code that uses the Printers collection, Forms collection, or Scale-mode-Constants.

- **Property *<object>.<property>* was not upgraded** Added when a property cannot be upgraded automatically. See Appendix A for a complete list of properties and their mappings from Visual Basic 6 to Visual Basic .NET.
- ***<objecttype> <object>* could not be resolved because it was within the generic namespace *<namespace>*** Added when code uses members of a variable declared as *Form* or *Control*. For information on fixing this type of soft-binding issue, see Chapter 10
- **Unload *<object>* was not upgraded** Added when *Unload* is used with a soft-bound form or control.
- **A string cannot be used to index the *<variablename>* control collection** In Visual Basic 6, all collections could be indexed by a name or by a numeric key. In Visual Basic .NET, only hash tables and the VB collection can be indexed by a name. Most other collections, such as the Windows Forms controls collection, have only a numeric key index.
- **Event parameter *<parameter name>* was not upgraded** Added when a form has an *Unload* or *QueryUnload* event. Windows Forms doesn't support the *Cancel* property for *Unload* or the *UnloadMode* property for *QueryUnload*.
- **Form property *<formname>*ScaleMode is not supported** In Visual Basic .NET, you can't set the scale mode at run time.
- ***<objecttype>* property *<variable>.<property>* is not supported at run time** Added when code sets a property that is read-only at run time.
- **ListBox|ComboBox property *<control>*.NewIndex was not upgraded** Added when code uses the *NewIndex* property of a ComboBox or ListBox control. *NewIndex* is not supported in Windows Forms.
- ***<control>* was upgraded to a Panel, and cannot be coerced to a PictureBox** PictureBoxes are upgraded to Panels if they contain child controls. This EWI is added when the Panel is then passed to a method that accepts a *PictureBox* parameter.

- ***<controltype>* Property *<control>.<property>* does not support custom mouse pointers** Added when code sets a custom mouse pointer. Custom mouse pointers are not automatically upgraded to Visual Basic .NET. You can rewrite your custom mouse pointer logic to use cursors. You can then load a cursor from a .cur file or a .resx file and set the cursor for a particular control.

Upgrade ToDos

- **Uncomment and change the following line to return the collection enumerator** Added to collection classes. The Upgrade Wizard upgrades most of the collection class code, but you need to finish the upgrade by adding the collection enumerator. In most cases, this means simply uncommenting one line.
- **Code was upgraded to use *<function>* which may not have the same behavior** Added to code that assigns a *ByteArray* to a string. In Visual Basic .NET, strings with an odd number of bytes will cause an exception because strings are stored in Unicode and therefore need two bytes per character. You should modify your code to ensure that the byte array is of an even length.
- "**Initialize" must be called to initialize instances of this structure** As we discussed previously, this EWI is added to types (upgraded to structures) when the type contains a fixed-size array. When declaring a variable of this type, you will need to call the *Initialize* method to initialize the array.
- **Add a delegate for *AddressOf <methodname>*** Added to code that uses *AddressOf*. See Chapter 11for more information on using *AddressOf* in Visual Basic .NET.
- ***LoadResStrings* method may need to be replaced** The Visual Basic 6 AppWizard often generated a *LoadResStrings* function. This function is not automatically upgraded to Visual Basic .NET. If your code uses the unmodified AppWizard function, you can replace it with the suggested code in Help.

Upgrade Warnings

- **As *<variable type>* was removed from *ReDim <variable>* statement** Arrays can be redimensioned using the *ReDim* statement only as the type they were originally defined as.
- **Arrays can't be declared with *New*** You can't declare arrays of classes with *New*. Variable declarations of the form *Dim x(10) As New Class1* are not allowed in Visual Basic .NET. The workaround is to initialize the classes when they are declared or as they are needed.
- **Structure *<variable>* may require marshalling attributes to be passed as an argument in this Declare statement** Added to APIs when a structure is being passed. For information on marshalling attributes, see Chapter 11.
- **Arrays in structure *<structure variable name>* may need to be initialized before they can be used** If a COM interface defines a structure with a fixed-size array, the array might need to be initialized before it can be used.
- **Array 'xxx' may need to have individual elements initialized** If an array of structures has been declared or redimensioned using *ReDim*, and the structure contains a fixed-length array, you will need to call *Initialize* for each member of the array.
- **Lower bound of array *<variable>* was changed from *<lowerbound>* to 0** Arrays in Visual Basic .NET must have a lower bound of 0. If the array has a lower bound of anything other than zero, it is changed to 0 and the wizard generates this EWI.
- **Can't resolve the name of control *<controlname>*** When the Upgrade Wizard needs to resolve the name of a control within a control array and it can't because the index is a variable, this warning is added.
- ***ParamArray <parameter>* was changed from *ByRef* to *ByVal*** In Visual Basic .NET, *ParamArrays* can be passed only by *ByVal*.

- **Use of Null/IsNull detected** *Null* is not supported in Visual Basic .NET, so it is upgraded to the closest equivalent, *System.DBNull.Value*. *IsNull* is changed to *IsDBNull*. You can't rely on *DBNull* having the same behavior as *Null* in Visual Basic 6. Also, Visual Basic 6 functions that accepted *Null* as a parameter and could return *Null*—such as *Left* and *Right*—now work only with strings. *Null* propagation is no longer supported in Visual Basic .NET. If *Null* is used in arithmetic expression, it will cause a run-time exception.

- ***<functionname>* has a new behavior** Some functions have different behaviors in Visual Basic .NET, and this EWI warns you of the difference. Here are two of the key differences:

  IsObject(<object>) is upgraded to *IsReference(<object>)* and, unlike its behavior in Visual Basic 6, returns *True* if the object is empty and *True* if the variable is a string. The *Dir* function no longer returns the directory entries . and ..

- **Class instancing was changed from *<instancing type>* to public** Single-use classes are changed to public classes during an upgrade.

- ***Sub Main* won't be called at component startup** If a DLL has a *Sub Main* in it, it won't be called when the first class is created.

- **Application will terminate when *Sub Main()* finishes** In Visual Basic 6, the application finishes when the *End* statement is called, or when all forms and objects are unloaded or destroyed. In Visual Basic .NET, the application ends when the *End* statement is called or when the startup object stops running. If your application has *Sub Main* as its startup object, the application will end when *Sub Main* ends. Several workarounds exist, including the use of *System.Windows.Forms.Application.Run* to create forms from within *Sub Main*. Forms loaded in this manner keep the application running once *Sub Main* finishes.

- ***<object>* event *<variable>.<event>* was not upgraded** Some events can't be upgraded to Visual Basic .NET. These events are changed into procedures and won't be fired as they were in Visual Basic 6. You have to call them yourself if you want to run the code. This applies to the *OLEDrag* and *OLEDragOver* events and to all *Font* events.

- **Event *<object>.<event>* may fire when form is initialized** Some events will be fired as the component is initialized. For example, if a check box has a default state of being selected, when the form is initialized, the *CheckBox.CheckStateChanged* event will fire. This leads to subtle differences if your event code refers to other controls that aren't fully created when this event is fired. You should modify your code to cope with this new behavior. This EWI is added for the following events:

OptionButton.Click	upgraded to *CheckChanged*
Form.Resize	upgraded to *Resize*
TextBox.Change	upgraded to *TextChanged*
ComboBox.Change	upgraded to *TextChanged*
CheckBox.Click	upgraded to *CheckStateChanged*
ComboBox.Click	upgraded to *SelectedIndexChanged*
DriveListBox.Click	upgraded to *SelectedIndexChanged*
ListBox.Click	upgraded to *SelectedIndexChanged*

- **Value *<value>* for *<controltype>* property *<control>.<property>* could not be resolved** Some properties and property enumerators have changed from Visual Basic 6 to Visual Basic .NET. This EWI is generated if the Upgrade Wizard can't map a property or property enumerator. It usually occurs because the property is being assigned to a variable or return value of another function that can't be resolved.
- **Couldn't resolve default property of object '*<objectname>*'** In some projects, this is the most common EWI you will see. It is added whenever a late-bound variable is set to a value. The Upgrade Wizard can't determine whether the assignment is to the variable or to the default property of the variable. Because it is late-bound, the wizard knows nothing about the variable and therefore generates this EWI.

- ***<function>* parameter '*<parameter>*' is not supported, and was removed** The *Wait* parameter of *AppActivate*, the *HelpFile* and *Context* parameters of *MsgBox*, and *InputBox* are not supported in Visual Basic .NET. These parameters are removed during an upgrade.
- **Assignment not supported: *KeyAscii* to a non-zero value** In Visual Basic .NET, the keypress parameter *KeyAscii* (now *KeyPressEventArgs.KeyChar*) can't be changed. If you have code that sets *KeyAscii* to 0, this is upgraded to *KeyPressEventArgs.Handled = True*. Code that sets it to any value other than 0 is marked with this warning.
- **Timer property *<controlname>*.Interval cannot have value of 0** *Timer.Interval = 0* deactivated a timer control in Visual Basic 6. In Visual Basic .NET, use the *Enabled* property to enable or disable the timer. Setting *Interval* to 0 causes an exception.

Design Errors

Design errors apply to forms and controls and are usually inserted only into the upgrade report. We've noted the EWIs that can be inserted into code.

- ***<controltype>* control *<controlname>* was not upgraded** This EWI is added to the report for controls that aren't supported in Windows Forms. The control is replaced with a red label placeholder. This EWI will occur with the following controls: OLE Container, Non horizontal/vertical lines, DAO Data, RDO Data, UpDown ActiveX, and the Shape control with the shape set to *Oval* or *Circle*.
- ***<controltype>* property *<controlname>.<propertyname>* was not upgraded** Added when a property or method does not map from Visual Basic 6 to Visual Basic .NET. This EWI is also added to code if the property or method is used at run time.
- ***<controltype>* property *<controlname>.<propertyname>* has a new behavior** Added to the report when a property, method, or event behaves differently in Visual Basic .NET. This issue is added for the following properties, methods, and events:
  - *Control.Keypress* event
  - *ComboBox.Change* event

- *Form.Activate* event
- *Form.Deactivate* event
- *Form.Picture* property
- *Form.QueryUnload* event
- *Form.Terminate* event
- *Form.Unload* event
- *Form.ZOrder* property
- *HScrollBar.Max* property
- *Image.Stretch* property
- *ListBox.Columns* property
- *ListBox.ItemCheck* event
- *OptionButton.Click* event
- *PictureBox.AutoSize* property
- *Screen.MousePointer* property
- *TextBox.TextLength* property
- *UserControl.Picture* property
- *VScrollBar.Max* property

- **PictureBox property <*controlname*>. Picture will be tiled** Added for a PictureBox with child controls. If a PictureBox contains child controls, it is upgraded to a panel because PictureBoxes in Windows Forms are not control containers. The background image of a Panel is always tiled.
- **ListBox|ComboBox property <*control*>. ItemData was not upgraded** Added when ListBoxes and ComboBoxes have *ItemData* set at design time. The *ItemData* property is not upgraded. (Note that run-time use of *ItemData* is upgraded.) The solution is to set the *ItemData* information for the ListBox or ComboBox at run time.
- **Only TrueType and OpenType fonts are supported in Windows Forms** Windows Forms doesn't support raster fonts. If a control has its *Font* property set to a raster font, the font is changed to the

default Windows Forms font (8-point Microsoft Sans Serif). This issue doesn't apply to ActiveX controls, which maintain their own fonts.

Global Warnings

- ***<objectname>* is not a valid Visual Basic 6.0 file and was not upgraded** The project item (form, module, or other item) has an invalid file format—perhaps it is corrupt or a Visual Basic 4 file. This file is ignored during the upgrade.
- **Could not load referenced component *<reference>*. It is recommended you install Visual Basic 6.0, with all referenced components, and ensure the application compiles and runs before upgrading.** Added when a reference, such as an ActiveX control or class library, cannot be found. The best way to avoid this issue is to ensure that the project runs on the machine before upgrading. The Upgrade Wizard needs to load all your project's references to examine the type libraries, create wrappers, and instantiate ActiveX controls. If this problem occurs, the upgrade is canceled.
- **Could not find file: *<projectitemtype>* *<filename>*. Please make sure this file is present before upgrading this project.** Added when a file can't be found. This warning can be issued if a file is located on a network share that is not available. If this problem occurs, the upgrade is canceled.
- ***<projectitemtype>* *<filename>* is a Visual Basic 5.0 file. For best results, upgrade the *<projectitemtype>* to Visual Basic 6.0 before upgrading it to Visual Basic. NET.** Added to the upgrade report whenever a project with a Visual Basic 5 form, class, and so on is upgraded.
- **The referenced component *<reference>* is missing a design time license** Added when the design-time license for a control is not installed. This warning usually occurs because a control is installed as part of a setup, without its design-time license. The Upgrade Wizard needs the design-time license because it instantiates ActiveX controls during the upgrade so that they can retrieve their properties for a Visual Basic 6 form. The Visual Basic .NET installation includes a registry file with the licenses for all the Windows Common controls. These licenses are stored in a registry file in the Extras directory; they are not installed with Visual Basic .NET.

- **The referenced project *<projectname>* is either not compiled or out of date. Please re-compile the project.** Occurs when a project references another project that hasn't been compiled. The solution is to compile the referenced project to a DLL. If this problem occurs, the upgrade is canceled.
- **MTS/COM+ objects were not upgraded** If a project has references to the Microsoft Transaction Server Type Library or the COM+ Services Type Library, they are removed during upgrade. These references will not work in Visual Basic .NET and should be replaced with .NET COM+ services objects. See Chapter 16 for more details.
- **The following line couldn't be parsed** Added to a code module when a line could not be upgraded because of syntax errors. The line is copied as is into the Visual Basic .NET module and marked with this EWI.
- ***<objecttype>* *<objectname>* was not upgraded** Applies to ActiveX documents and property pages. These project items can't be automatically upgraded.

Upgrade Notes

- **There may be differences in databinding behavior** Added to the upgrade report whenever an application has ADO data binding. The warning has a hyperlink to a Help topic that gives general advice about fixing issues with ADO data binding.
- **Def *<variabletype>* statement was removed. Variables were explicitly declared as type *<type>*.** Added to the declaration section of modules that had statements such as *DefInt* and *DefStr*. The variables are changed to be explicitly declared. This message is purely informative.
- ***<variable>* was changed from a Constant to a Variable** A constant was changed to a variable because the data type can't be stored as a constant in Visual Basic .NET. This message is added for constants declared as colors.
- ***NewEnum* property was commented out** When collection classes are upgraded, the Upgrade Wizard comments out the old *NewEnum* property and inserts a ToDo comment describing how to implement the collection enumerator.

- **Object <*variable*> may not be destroyed until it is garbage collected** Added to code that sets an object to *Nothing*. Objects set to *Nothing* aren't actually destroyed until the .NET Garbage Collector performs its next collection.
- ***IsMissing*(<*variable*>) was changed to *IsNothing*(<*variable*>)** The *IsMissing* function was replaced with *IsNothing*.
- **#If #EndIf block was not upgraded because the expression <*expr*> did not evaluate to True or was not evaluated** If your code has conditional compilation statements, code in the *False* branch will not be upgraded. The Visual Basic 6 code will be copied as is into your upgraded application.
- **Remove the next line of code to stop form from automatically showing** Added to multiple document interface (MDI) child forms when the Visual Basic 6 MDIForm has the property *AutoShowChildren = True*. The Upgrade Wizard inserts the line *Me.Show* into the child form to give it the same behavior as in Visual Basic 6.
- **The following line was commented to give the same effect as VB6** Code that does nothing in Visual Basic 6—but would have an effect in Visual Basic .NET—is commented out. This applies to code that sets *Form.BorderStyle* at run time.
- **<*control*>.<*event*> was changed from an event to a procedure** Added to these events: *HScrollBar.Change*, *HScrollBar.Scroll*, *VScrollBar.Change*, *VScrollBar.Scroll*, and *MenuItem.Click*. The event was changed to a procedure and called from another event. This change is made because Windows Forms combines some events that used to be separate events into a single event.
- **<*statement*> was upgraded to <*statement*>** Added when a statement is upgraded to a corresponding statement with a different name. This note is added for the *Erase* statement, which is upgraded to *System.Array.Clear*.

Which Problems Are Not Detected?

Perhaps you're asking yourself, "Which errors doesn't the Upgrade Wizard catch?" The Upgrade Wizard doesn't warn you about five important errors:

- The wizard doesn't warn about the *New* statement. The following code has a different behavior in Visual Basic 6 than in Visual Basic .NET (see Chapter 1for a full explanation):

```
Dim x As New y
```

- The wizard doesn't output an EWI for this because in 99.99 percent of the cases you won't notice the difference. If the wizard did create a warning for every occurrence, some projects would be overcome with EWIs.
- The wizard doesn't detect all coercion problems (in which a variable of one type is assigned to another type). Some coercions are allowed in Visual Basic 6 but not in Visual Basic .NET. Generally these errors are easy to fix, since they almost always result in compile errors.
- It is common to pass variables as parameters to functions. Often these parameters are defined as *ByRef*, which means that if the function changes the value of the parameter, the underlying variable also changes. For example, in the following code snippet, *myVariable* is passed to a function as *myParameter*, and the function changes *myParameter* to 5, which also changes *myVariable* to 5, since it is passed as a *ByRef* parameter.

```
Sub Main()
   Dim myVariable As Integer
   myFunction myVariable
   MsgBox myVariable
End Sub

Function myFunction(myParameter As Integer)
   myParameter = 5
End Function
```

- Visual Basic 6 has one exception to this behavior: if you pass a property as a *ByRef* parameter, it will not be changed. For example, suppose that our application has a form with a TextBox on it called Text1. The following line of code passes the *Text* property of Text1 to the function *myFunction*.

```
myFunction Form1.Text1.Text
```

- In Visual Basic 6, because the *Text* property is a property, it is not changed when the function explicitly changes the value. In effect, properties are always passed *ByVal*. Visual Basic .NET does not have this limitation; properties passed as *ByRef* parameters are treated as *ByRef* parameters. If the example just given was upgraded to Visual Basic .NET, the *Text* property would be changed.
- In some cases, this may lead to subtle differences in your project's run-time behavior. You can force Visual Basic .NET to pass properties as *ByVal* parameters by adding parentheses around the variable. For example, this code passes the Text property *ByRef*:

  ```
  myFunction(Me.Text1.Text)
  ```

  whereas this code passes the Text property *ByVal* (just as Visual Basic 6 did):

  ```
  myFunction((Me.Text1.Text))
  ```

- In Visual Basic 6, you can use *ReDim* to redimension a variable to have a different number of indexes. For example, the following code is valid in Visual Basic 6 but causes a compile error in Visual Basic .NET:

  ```
  Dim x()
  ReDim x(5, 5)
  ```

- The Upgrade Wizard doesn't warn you about this, but the compile error makes it easy to detect and easy to fix. In Visual Basic .NET, each time a variable is redimensioned, it must always have the same number of indexes. In Visual Basic .NET, if you change the *Dim* statement to dimension the variable with two indexes (using a comma), it works perfectly:

  ```
  Dim x(,)
  ReDim x(5, 5)
  ```

- The wizard warns you about problems it detects with ActiveX controls. However, there are subtle differences between hosting ActiveX controls in Visual Basic 6 and hosting them in Visual Basic .NET. The wizard doesn't know about all these differences, and therefore it can't warn you. For a discussion of using ActiveX controls in Visual Basic .NET, see Chapter 9 and Chapter 13.

It's important to be conscious of these issues. Learn the methods that will allow you to handle each of them just as you handle issues that the Upgrade Wizard detects.

Conclusion

EWIs are a valuable part of your upgrade. They alert you to issues in your code and show you how to fix the problems. In a perfect world we wouldn't need them—everything would upgrade without problems. The next best thing is to be aware of problems and know how to fix them. Looking over the list, you'll see that many errors can be avoided entirely by writing your Visual Basic 6 code in a way that prepares it for Visual Basic .NET. Doing a two-pass upgrade is a great way to see what types of EWIs your Visual Basic 6 program will have when it is upgraded to Visual Basic .NET.