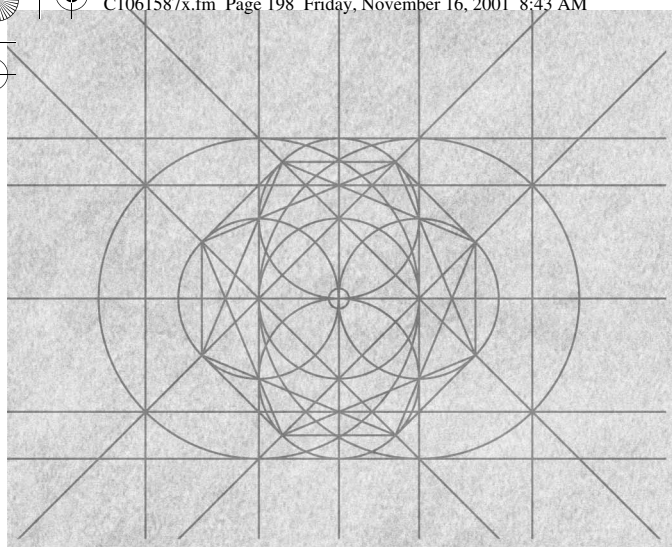
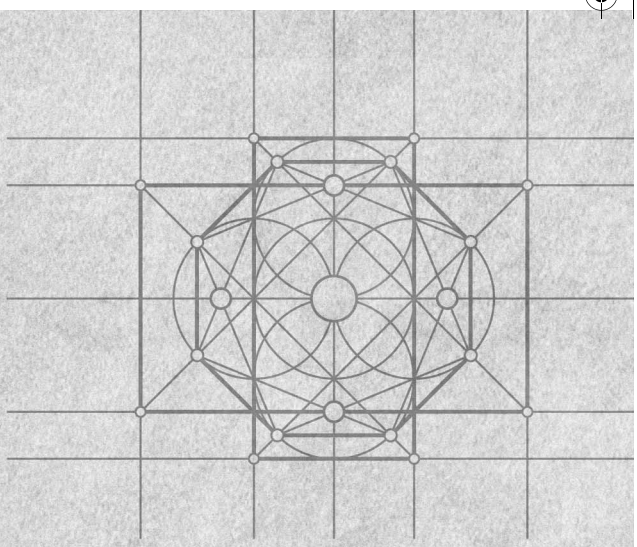
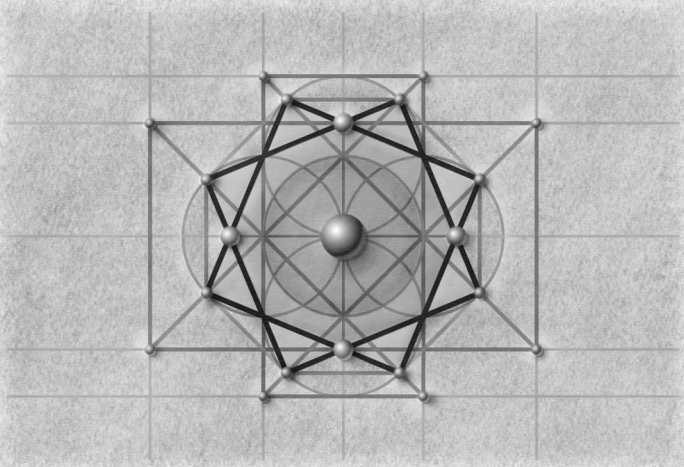
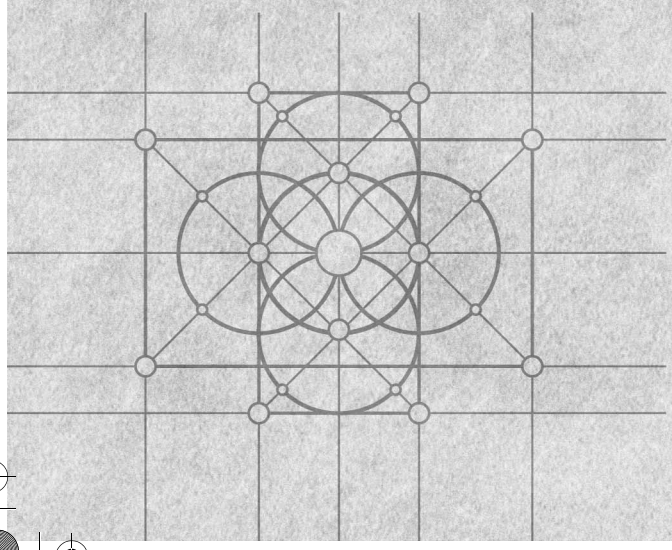
Part III

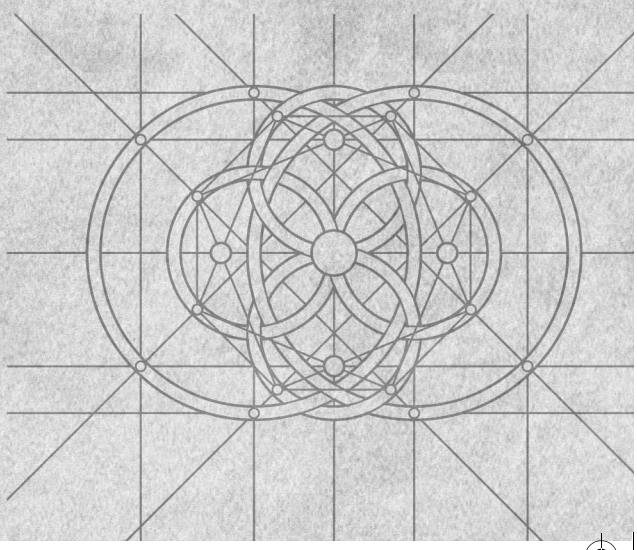
Getting Your Project Working

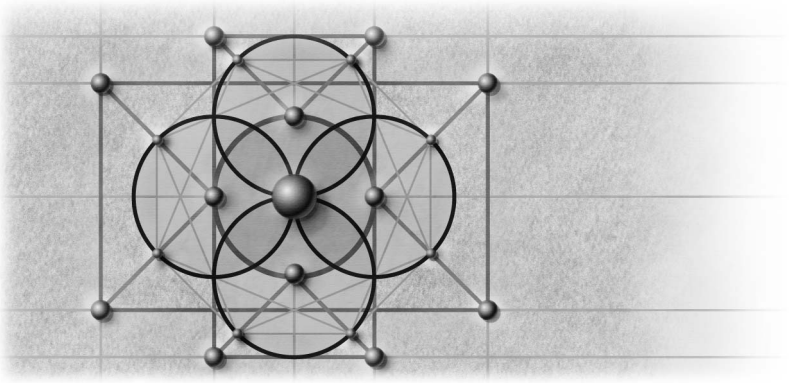












10

Ten Common Upgrade Problems

Chapter 4 looked at common issues with Microsoft Visual Basic 6 code and discussed how you could resolve them before upgrading. This chapter also focuses on resolving common upgrade issues—but this time we look at the issues found after the wizard has finished upgrading your project. To this end, we describe ten very common upgrade problems and how to fix them. Some of these problems are due to language differences and new control behaviors. Others are related to the different run-time behavior of the .NET Framework. You'll learn how to avoid these issues in the first place and how to resolve them if you encounter them after you begin upgrading. You'll also find sample applications on the companion CD that illustrate some of the situations discussed here.

Default Properties

Default properties are no longer supported in the common language runtime. If the Upgrade Wizard can determine a default property, it will properly modify your code to state the property explicitly in Visual Basic .NET. If it cannot determine the property being referenced (in the case of late binding), the wizard will leave your code as is and insert a warning highlighting the issue for you to resolve. In those cases, the offending line of code will be preserved through the upgrade and tagged to bring it to your attention. While it is possible (and often more desirable) to go back to your original code and modify it, there are definitely cases in which it is not possible to make those changes anywhere but in the upgraded application.

The following code samples demonstrate the differences in upgrade results when default properties are accessed using both late-bound and early-bound objects. The code is contained in the DefaultProperties sample included on the companion CD.

```
Private Sub CopyButton_Click()
   EarlyBoundCopy Text1, Text2
   LateBoundCopy Text1, Text3
End Sub

' This method's parameters are explicitly defined as TextBoxes
Private Sub EarlyBoundCopy(sourceCtrl As TextBox, destCtrl As TextBox)
   destCtrl.Text = sourceCtrl
End Sub

' This method's parameters are variants (the default type)
Private Sub LateBoundCopy(sourceCtrl, destCtrl)
   destCtrl.Text = sourceCtrl
End Sub
```

Although this code runs just fine in Visual Basic 6, the Upgrade Wizard will insert two run-time warnings in your code. This type of warning means that the code will compile after upgrading but will generate an exception at run time. Specifically, the *LateBoundCopy* method will generate an *InvalidCast-Exception*. The exception says the following: "Cast from type 'TextBox' to type 'String' is not valid."

Here is the code the Upgrade Wizard generates:

```
Private Sub CopyButton_Click(ByVal eventSender As System.Object, _
   ByVal eventArgs As System.EventArgs) Handles CopyButton.Click
   EarlyBoundCopy(Text1, Text2)
   LateBoundCopy(Text1, Text3)
End Sub

' This method's parameters are explicitly defined as TextBoxes
Private Sub EarlyBoundCopy(ByRef sourceCtrl As  _
   System.Windows.Forms.TextBox, ByRef destCtrl As _
   System.Windows.Forms.TextBox)
   destCtrl.Text = sourceCtrl.Text
End Sub

' This method's parameters are variants (the default type)
Private Sub LateBoundCopy(ByRef sourceCtrl As Object, _
   ByRef destCtrl As Object)
   'UPGRADE_WARNING: Couldn't resolve default property of object
   'destCtrl.Text.
   'UPGRADE_WARNING: Couldn't resolve default property of object
   'sourceCtrl.
   destCtrl.Text = sourceCtrl
End Sub
```

When you encounter these kinds of problems, you will need to modify your code. Two options are available. First, you can change the definition of the variable to be strongly typed in Visual Basic 6 and run your application through the Upgrade Wizard again. Your other option is to explicitly specify the desired property in Visual Basic .NET. This last option results in the *LateBoundCopy* method looking like this:

```
Private Sub LateBoundCopy(ByRef sourceCtrl As Object, _
   ByRef destCtrl As Object)
   destCtrl.Text = sourceCtrl.Text
End Sub
```

> **Note** The second option involves modifying your code to explicitly use the property you want to read or modify. Notice that this is exactly what the Upgrade Wizard does for you when it knows what type it is dealing with. This example illustrates the advantage of always using early binding. The Upgrade Wizard will know what the default properties are and will take care of the work for you. Using late binding puts you in the position of going through all of the upgrade warnings and manually entering the desired properties.

The fact that you can upgrade an application that contains these kinds of run-time problems should lead you to two conclusions. First, you need to pay close attention to the issues raised by the Upgrade Wizard. Second, you must test your upgraded application thoroughly to make sure that the original functionality has been maintained. The next section examines how using default properties can also lead to additional and more subtle run-time differences.

AddItem and *ToString* with COM Objects

In Visual Basic 6, the *AddItem* method is used to add a new entry to a ListBox or ComboBox control. In Visual Basic .NET, the equivalent operation involves calling the *Add* method of the Items collection for the ComboBox or ListBox Windows Forms controls. This method takes an object as a parameter, instead of a string. Because only strings can be displayed in a ListBox, the *Add* method implicitly calls the *ToString* method on the passed object. For variables that have a primitive value, such as a string or a number, the method returns the value as a string. For other objects, however, the *ToString* method's behavior varies significantly from class to class.

> **Note** Officially, Visual Basic .NET no longer supports implicit default properties. Unofficially, however, this is not universally true. All classes in the .NET Framework derive from *System.Object*. *System.Object* defines several methods, one of the most important of which is the *ToString* method. In some ways, *ToString* is emerging as a de facto default property. For instance, ComboBoxes display strings in their lists, yet the *ComboBox.Items.Add* method accepts an object, not a string. How does the *Add* method populate the list? It calls the object's *ToString* method with the expectation that the *ToString* method of any object returns the actual information that is desired.

Consider the following example of adding items to a ListBox from a TextBox in Visual Basic 6. This example is contained in the ListBoxExample sample project on the companion CD. It adds the contents of a TextBox control to a ListBox control. The key here is that we are relying on the default method of the TextBox control (*Text*) being called when the control is passed to the *AddItem* method.

```
Option Explicit

' Button Click Event Handler
Private Sub AddButton_Click()
   AddText Text1
End Sub

' Adding the control to the ListBox
Private Sub AddText(text)
   ' We are relying on the default property here
   List1.AddItem text
End Sub
```

The problem with this code is its use of the default property of the TextBox object when the object is late-bound. When a TextBox is used in this context, Visual Basic assumes that you want the default property specified in the COM object's type library. In this case, the *Text* property of the *TextBox* object is defined as its default property. This means that the following two lines of code are functionally equivalent:

```
List1.AddItem Text1
List1.AddItem Text1.Text
```

As we mentioned previously, the use of a *Variant* for handling the *TextBox* object in the *AddText* method prevents the Upgrade Wizard from resolving the default property. The following example shows what the code looks like after upgrading:

```
' Button Click Event Handler
Private Sub AddButton_Click(ByVal eventSender As System.Object, _
   ByVal eventArgs As System.EventArgs) Handles AddButton.Click
   AddText(Text1)
End Sub

' Adding the control to the ListBox
'UPGRADE_NOTE: Text was upgraded to text_Renamed.
Private Sub AddText(ByRef text_Renamed As Object)
   ' We are relying on the default property here
   'UPGRADE_WARNING: Couldn't resolve default property of
   'object text_Renamed.
   List1.Items.Add(text_Renamed)
End Sub
```

What happens? The code runs as is and does not generate an exception at run time. However, the TextBox's *ToString* method is called. Unfortunately, the TextBox does not return just the displayed text; it also prepends the string with its class path. So, for example, if the TextBox displays the string "Hello World," the *ToString* method returns "System.Windows.Forms.TextBox, Text: Hello World." To fix the problem, you need to modify your code to eliminate the use of default properties. Doing so would change the line in which the items are added to the ListBox to the following:

```
' This explicitly passes the contents of the Text property to the
' ListBox Add method.
List1.Items.Add(text_Renamed.Text)
```

It is important to emphasize that if the original sample had not used late binding, no modifications would have been required.

Deterministic Finalization and Garbage Collection

Garbage collection is a new concept to Windows platform development. It is quite a departure from the COM memory-management mechanism of reference counting and offers significant benefits. In COM, objects use reference counting to keep track of the number of active references to determine their lifetime. When an object's reference count reaches zero, the object is destroyed. Reference counting presents several problems within COM because of issues with

potential reference leaks or circular references. However, it is, for the most part, predictable. Reference counting makes it possible to determine when and where an object will be destroyed and, as a result, to manipulate the lifetime of the object in a deterministic way—hence the term **deterministic finalization**.

Visual Basic 6 made COM programming much simpler because it hid a great deal of the COM implementation details. But you could rely on Visual Basic's behavior with respect to COM object lifetimes. The following example illustrates how reference counting in Visual Basic 6 can influence an object's lifetime:

```
Dim x As Connection, y As Connection
Set x = new Connection 'Create an object and set refcount = 1
Set y = x       'set refcount = 2
Set x = Nothing 'set refcount = 1
Set y = Nothing 'set refcount = 0; and destroy the object
```

It is not necessary to explicitly dereference your objects in Visual Basic 6. When a variable goes out of scope, Visual Basic 6 automatically dereferences the object for you:

```
Sub MySub()
   Dim x As Connection, y As Connection
   Set x  = New Connection 'Create an object and set refcount = 1
   Set y = x   'set refcount = 2
End Sub
```

In Visual Basic .NET, the picture is drastically different. For the most part, it is no longer possible to determine the exact lifetime of a given object because Visual Basic .NET does not use reference counting. Instead, it handles memory management through garbage collection. Putting aside the implementation details, this change produces distinct behavioral differences that will have implications for your applications. First of all, setting a reference to *Nothing* will not cause that object to be immediately destroyed. Garbage collection guarantees that the object will be destroyed at some point, but there is no guarantee as to when it will be destroyed.

The effects of this change are best illustrated by the following code in Visual Basic .NET:

```
Dim conn As New ADODB.Connection
conn.Open( ... )
Dim rs = conn.Execute( ... )
conn = Nothing  ' The object is not destroyed here and the
                ' connection is still open
```

> **Note** You might ask why Microsoft chose to make this change from a deterministic memory-management scheme to a nondeterministic one such as garbage collection. The general consensus was that too much of COM programming was related to memory-management issues. Garbage collection frees the developer from many of these arduous tasks, enabling the developer to focus more on designing program logic then on memory management. It does require a slightly different programming approach when dealing with objects that represent physical system resources. For the most part, though, you don't really need to worry about it. Garbage collection is quite a positive change and should result in fewer problems with memory leaks in your applications.

In Visual Basic 6, we could rely on setting the *conn* reference to *Nothing* to close the connection and destroy the underlying *Connection* object. In Visual Basic .NET, setting *conn* to *Nothing* simply marks the object for collection. The underlying physical connection to the database remains open, consuming both memory and network resources. Thus, it is imperative when working with objects that represent real physical resources (such as files, graphics handles, and database connections) that you explicitly release those resources before you dereference the object (or allow it to go out of scope). A proper way to handle this would be as follows:

```
Dim conn As New ADODB.Connection
conn.Open( ... )
Dim rs = conn.Execute( ... )
conn.Close()
conn = Nothing
```

The *Close* method ensures that the database connection is released immediately. This is by far the best way to handle any physical resource. Whenever you deal with a physical resource, follow this general rule: open the resource as late as possible and release it as soon as possible. This practice will ensure the most efficient use of system resources.

Bringing a Little Determinism to the Party

It is possible to ensure that your objects are destroyed before your application proceeds. To do so, you need to cause a garbage collection to occur and then wait for the collection to complete. You can accomplish these tasks by calling the following two methods.

```
GC.Collect()
GC.WaitForPendingFinalizers()
```

It is pretty obvious that the first method causes a collection to occur. You need to realize, however, that the *Collect* method is an asynchronous call (hooray, free threading!), meaning that your application will not wait for the collection to complete before continuing. That is why it is important to call *WaitForPendingFinalizers*, which will block your application until the collection has completed.

Now that you've seen how forcing garbage collection can be done, here is why you should almost never do it: Garbage collection is, by definition, an expensive operation. For this reason, it takes place relatively infrequently, instead of whenever and wherever an object is dereferenced. If you have code that needs to work no matter what, then by all means take advantage of *Collect* and *WaitForPendingFinalizers*. Ultimately, however, this technique is just a quick fix. For the long term, you should probably investigate how to eliminate this dependency.

Generic Objects (*Control*/*Form*/*Screen*)

Generic objects are a form of soft binding (discussed in Chapter 4). A problem arises when you use methods such as *ActiveForm* or *ActiveControl* that return object types (*Form* or *Control*) that you use as though they were a more specific type. For example, suppose that you have created a form in your application. If you add a TextBox to that form (call it Text1), you have a distinct class derived from the *Form* object with a property (*Text1*) that does not exist in the parent *Form* class.

When you call the method *ActiveForm*, it returns a *Form* object that does not have a *Text1* property. In Visual Basic 6, you can use that strongly typed object in a late-bound fashion:

```
Screen.ActiveForm.Text1.Text = "This is a test"
```

Visual Basic 6 allows you to access the properties and methods of these objects. Visual Basic .NET, on the other hand, has more strict type requirements. While it allows both early and late binding, soft binding is no longer supported. When you upgrade this kind of code, you need to cast the result from *ActiveForm* to either an object (forcing late binding) or to the specific class that contains the method or property you are attempting to invoke (forcing early binding). The upgraded code can take one of the following forms:

```
' Forcing late binding by casting to Object
CType(ActiveForm, Object).Text1.Text = "This is a test"

' Forcing early binding by casting to Form1
CType(ActiveForm, Form1).Text1.Text = "This is a test"
```

Although it is up to you to decide what form you prefer, we recommend that you use early binding whenever possible. There is nothing intrinsically wrong with using late binding, but early binding provides the developer with a level of compile-time validation. Late binding can result in errors that are exposed only at run time. It is also slower than early binding because the run-time environment has to bind the methods and properties to the object at run time (hence the term *late binding*). In an informal test, early-bound property access was more than five times faster than late-bound access. (See the Visual Basic .NET project LateBinding Performance on the companion CD. As you run it, watch the Output window in the debugger.) Why make life more difficult for yourself when you can have the best of both worlds? Early binding is best, unless there is no feasible alternative.

Dim...As New

You will encounter upgrade issues with the *As New* syntax in two types of cases. The first is when you use *As New* to declare a variable. The second is when you use it to declare an array. The use of *As New* to declare a variable is still supported in Visual Basic .NET, but it no longer supports implicit declaration. Array declaration with *As New* is not supported at all. The following code shows how the Upgrade Wizard handles both types of declarations.

Here are the declarations in Visual Basic 6:

```
Dim x As New Class1
Dim y(5) As New Class2
```

Here's how the Upgrade Wizard handles the declarations:

```
Dim x As New Class1()
'UPGRADE_WARNING: Arrays can't be declared with New.
Dim y(5) As New Class2()
```

Notice that the Upgrade Wizard leaves the array declaration alone. When you first upgrade the project, there will be a compiler error on this line because *As New* is no longer supported for arrays. The best way to deal with this situation is fairly simple: initialize your array in a loop.

```
Dim y(5) As Class2
For i = 0 To 5
   y(i) = New Class2()
Next
```

This solution does not address the other side of the *As New* behavior: implicit instantiation. For example, the following code will fail in Visual Basic .NET with a *NullReferenceException* at run time:

```
Dim x As New Object1()
x.AnyMethod()
x = Nothing
x.AnyMethod() ' This will cause a NullReferenceException
```

The Upgrade Wizard will help you isolate these problems. First it will insert a warning wherever you have declared an array with *As New*. It will also generate another warning wherever you set an object to *Nothing*. You can use these warnings to track down spots in your code where you might run into difficulties. The only way to resolve these problems is to explicitly create a new instance of *Object1* or to test for *Nothing* whenever you access a variable that might have been set to *Nothing*.

```
Dim x As New Object1()
x.AnyMethod()
x = Nothing
x = New Object1()
x.AnyMethod()
```

As New is an example of a language feature that differs significantly in functionality from Visual Basic 6 to Visual Basic .NET. It can present a host of issues in an application that relies too closely on its previous behavior. Watch yourself.

Sub Main (or Default Form)

In Visual Basic .NET, your application's lifetime is now defined differently than it was in Visual Basic 6. When the startup object finishes, the application ends. In Visual Basic 6, an application remained active as long as there were dialog boxes or windows open. This represents a change in how your application's life cycle is determined. For the most part, however, you probably will not have to worry about this change, unless your application closes its main form but expects to continue running. The application model assumes the use of a main form that runs on the main thread. When that form terminates, your application terminates by default.

There are two main startup scenarios for Visual Basic applications. The first is when the startup object is a form. The second is when the startup object is a *Sub Main* method. *Sub Main* is typically used in two situations. You may implement a *Sub Main* method if you need to customize the loading process or if you used the Application Wizard to create your application.

> **Note** The Upgrade Wizard attempts to treat Application Wizard projects as a special case, but if the *Sub Main* method is too heavily modified, it will not be able to do much for you. See Chapter 17for more on upgrading VB Application Wizard projects.

As a general rule, when you are opening a form from *Sub Main*, use the *System.Windows.Forms.Application.Run* method to start your application. This method begins running a standard Windows application messaging loop on the current thread and makes the specified form visible. If you're creating another form that may replace the original form, you should create it on a separate thread to ensure that your application does not terminate prematurely. Otherwise, when the specified form is closed, the application will terminate immediately. The following example demonstrates the use of the *Application.Run* method in a fictional *Sub Main*.

```
' This is how to start your application from Sub Main
System.Windows.Forms.Application.Run(New Form1())
```

Font Disparities

In Visual Basic 6, it is possible to use two different types of fonts: raster and TrueType. Although TrueType fonts will upgrade to Visual Basic .NET, raster fonts are no longer supported. This change can present a problem for your application. What happens to your carefully laid-out forms? Table 10-1 shows how the Upgrade Wizard translates various Visual Basic 6 fonts during the upgrade process.

Table 10-1 Visual Basic Fonts and How They Are Upgraded

Font Name	Type	Converted Font	Style Preserved?
Arial	TrueType	Arial	Yes
Comic Sans MS	TrueType	Comic Sans MS	Yes
Courier	Raster	Microsoft Sans Serif	No: 8pt Plain
Courier New	TrueType	Courier New	Yes
FixedSys	Raster	Microsoft Sans Serif	No: 8pt Plain
Microsoft Sans Serif	TrueType	Microsoft Sans Serif	Yes
Modern	Raster	Microsoft Sans Serif	No: 8pt Plain
MS Sans Serif	Raster	Microsoft Sans Serif	Yes
MS Serif	Raster	Microsoft Sans Serif	Yes
Roman	Raster	Microsoft Sans Serif	No: 8pt Plain
Script	Raster	Microsoft Sans Serif	No: 8pt Plain
Small Fonts	Raster	Microsoft Sans Serif	No: 8pt Plain
System	Raster	Microsoft Sans Serif	No: 8pt Plain
Terminal	Raster	Microsoft Sans Serif	No: 8pt Plain

If you used the default font MS Sans Serif in a Visual Basic 6 form, your formatting will be preserved in Visual Basic .NET but the font will be changed to the Microsoft Sans Serif TrueType font. You may encounter a problem with layout disparities owing to the slight differences between the two fonts. Controls that are laid out with very tight constraints on your forms might exhibit a wrapping effect.

Non-TrueType fonts will lose their formatting when upgraded. The best way to ensure that your application preserves its look is to modify your Visual Basic 6 application to use TrueType fonts, thus ensuring that your upgraded forms will require minimal control repositioning. Figures 10-1 and 10-2 demonstrate what happens when raster fonts are upgraded. Figure 10-1 shows a sample application (FontTest, included on the companion CD) prepared in Visual Basic 6. Notice how both raster and TrueType fonts are represented.

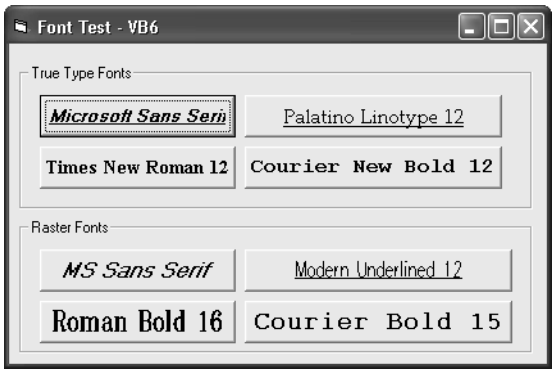


Figure 10-1 Raster and TrueType fonts in Visual Basic 6.

As Figure 10-2 shows, the fonts can change dramatically after the application is upgraded to Visual Basic .NET with the Upgrade Wizard. This example demonstrates some of the challenges you will face in your form layouts.

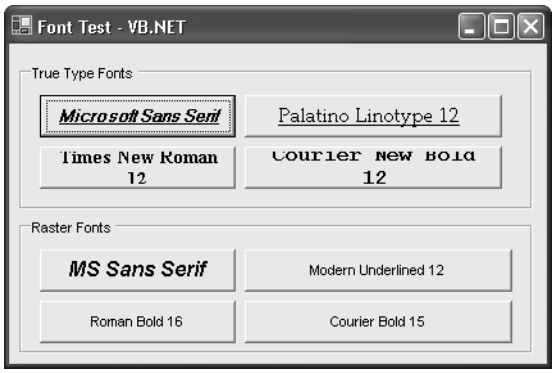


Figure 10-2 Raster and TrueType fonts after upgrading.

Notice that all the raster fonts, with the exception of MS Sans Serif, lost their formatting and ended up in plain 8-point Arial. As you can see, you have no choice but to fix your fonts manually. Stick with MS Sans Serif, and you should have few problems. (With luck, only spot fixes will be required.) The other choice is to use TrueType fonts, which will always maintain formatting, although they too may require some tweaking.

Bad Constants

In Visual Basic 6, you can create a set of named constants, known as an *enumeration*. Each member of the enumeration is assigned an integer value; in code, the member name evaluates to its assigned integer. Because enumerations in Visual Basic 6 represent integer values instead of distinct types, the process can verify only their underlying values. Unfortunately, this arrangement allows developers to use constants intended for different purposes interchangeably and also to use simple integers (representing the underlying values) in place of the constants. Visual Basic .NET is more strict and requires the correct enumeration constants because enumerations are now actual types, not just collections of friendly names for integers. The most common example of the use of bad constants is in relation to the *Screen.MousePointer* property. Constants such as *vbNormal* are frequently used in place of *vbDefault* (a proper constant for the *MousePointer* property). It is also fairly common for developers to use the underlying values for constants. The following example from Visual Basic 6 demonstrates three equivalent statements that work just fine:

```
' Correct MousePointer constant
Screen.MousePointer = vbDefault

' Incorrect constant, same underlying value
Screen.MousePointer = vbNormal

' The same underlying value
Screen.MousePointer = 0
```

When you upgrade your project, however, the Upgrade Wizard will not be able to upgrade the *vbNormal* constant because it is not defined for the *MousePointer* and you will be left with an upgrade warning and a compile error. On the other hand, the Upgrade Wizard can work with an underlying constant, if used, and can translate it to a proper constant in Visual Basic .NET. The following results from the Upgrade Wizard illustrate this scenario:

```
' Correct MousePointer constant
'UPGRADE_WARNING: VB.Screen property Screen.MousePointer has a
'new behavior.
System.Windows.Forms.Cursor.Current = _
   System.Windows.Forms.Cursors.Default

' Incorrect constant, same underlying value
'UPGRADE_ISSUE: Unable to determine which constant to upgrade
'vbNormal to.
'UPGRADE_ISSUE: VB.Screen property Screen.MousePointer does not
```

```
'support custom mousepointers.
'UPGRADE_WARNING: VB.Screen property Screen.MousePointer has a
'new behavior.
System.Windows.Forms.Cursor.Current = vbNormal

' The same underlying value
'UPGRADE_WARNING: VB.Screen property Screen.MousePointer has a
'new behavior.
System.Windows.Forms.Cursor.Current = _
   System.Windows.Forms.Cursors.Default
```

If you have used an incorrect constant, you will need to fix it on your own. The Upgrade Wizard cannot help you, other than highlighting the problem in the upgrade report. Thankfully, the solution is very simple, thanks to IntelliSense. Simply delete the offending constant, and you get an IntelliSense menu listing the proper constant options. Pick the appropriate constant, and you are on your way.

Drag and Drop

Drag-and-drop capability is an important function of most modern applications. There are a number of important differences between Visual Basic 6 and Visual Basic .NET with respect to drag and drop. To better understand how these changes will affect your application, let's take a quick overview of this feature.

Drag and Drop in Visual Basic 6

Visual Basic 6 supported two kinds of drag-and-drop operations. Standard drag and drop is intended to support drag and drop between controls within a single form. OLE drag and drop was designed to support drag-and-drop capability between applications and is the focus of this section.

The standard Visual Basic 6 controls have varying degrees of support for OLE drag-and-drop operations. Table 10-2 lists the standard Visual Basic 6 controls and their support for manual and automatic drag-and-drop operations.

Table 10-2 Visual Basic 6 Controls Grouped According to Their Support for OLE Drag and Drop

Controls	OLEDragMode	OLEDropMode
TextBox, PictureBox, Image, RichTextBox, MaskedBox	*VbManual, vbAutomatic*	*vbNone, vbManual, vbAutomatic*
ComboBox, ListBox, DirListBox, FileListBox, DBCombo, DBList, TreeView, ListView, ImageCombo, DataList, DataCombo	*VbManual, vbAutomatic*	*vbNone, vbManual*
Form, Label, Frame, CommandButton, DriveListBox, Data, MSFlexGrid, SSTab, TabStrip, Toolbar, StatusBar, ProgressBar, Slider, Animation, UpDown, MonthView, DateTimePicker, CoolBar	Not supported	*vbNone, vbManual*

For the sake of brevity, this section deals exclusively with a manual drag-and-drop application. To better illustrate how the upgrade process affects OLE drag-and-drop operations, we have provided a sample application named OLEDragAndDrop on the companion CD. It permits the user to drag and drop image files to and from the application and implements OLE drag and drop in full manual mode. This sample should give you a better idea of exactly what changes you will need to make in your own application. Figure 10-3 shows an image from the sample OLEDragAndDrop application.

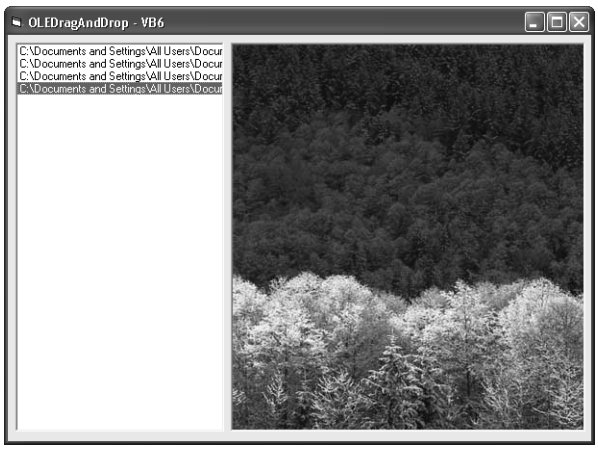


Figure 10-3 OLEDragAndDrop application.

Figure 10-4 outlines the life cycle of an OLE drag-and-drop operation in Visual Basic 6. The next section moves on to how things are done in Visual Basic .NET.

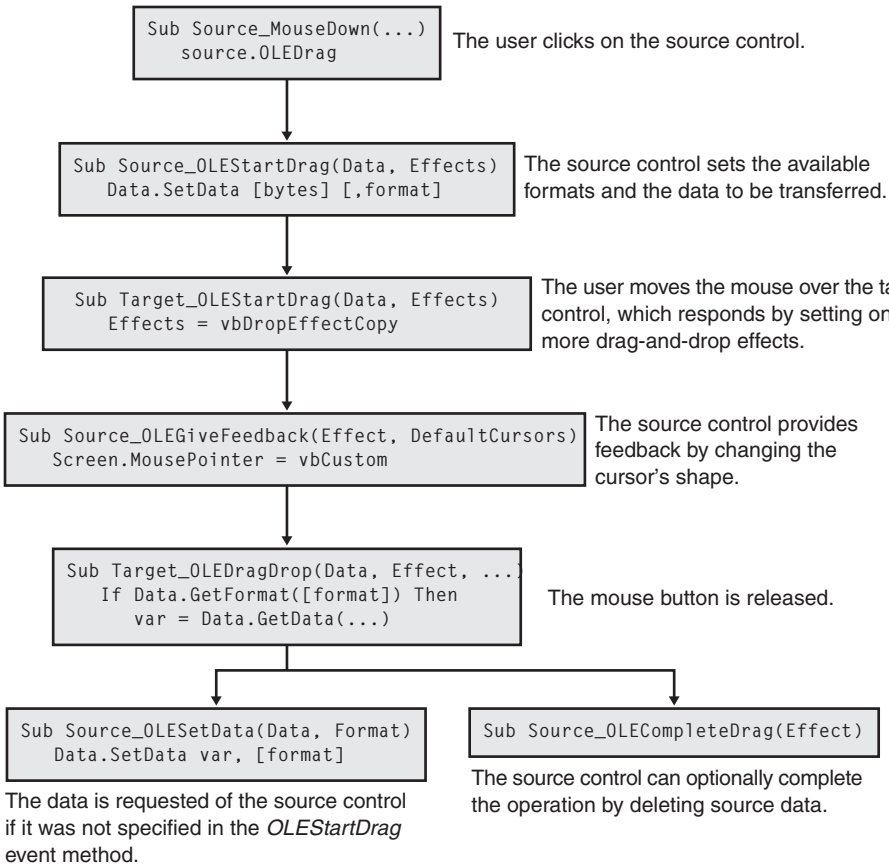


Figure 10-4 Life cycle of a Visual Basic 6 drag-and-drop operation.

Drag and Drop in Visual Basic .NET

In Visual Basic .NET, the drag-and-drop operations have been consolidated into a single framework. Drag and drop between controls is handled in exactly the same manner as drag and drop between applications. This change simplifies the programming burden for new development but requires the rewriting of drag-and-drop code for existing applications. Figure 10-5 outlines the life cycle of a drag-and-drop procedure in Visual Basic .NET. Note that this cycle applies to both control-to-control and application-to-application drag and drop.

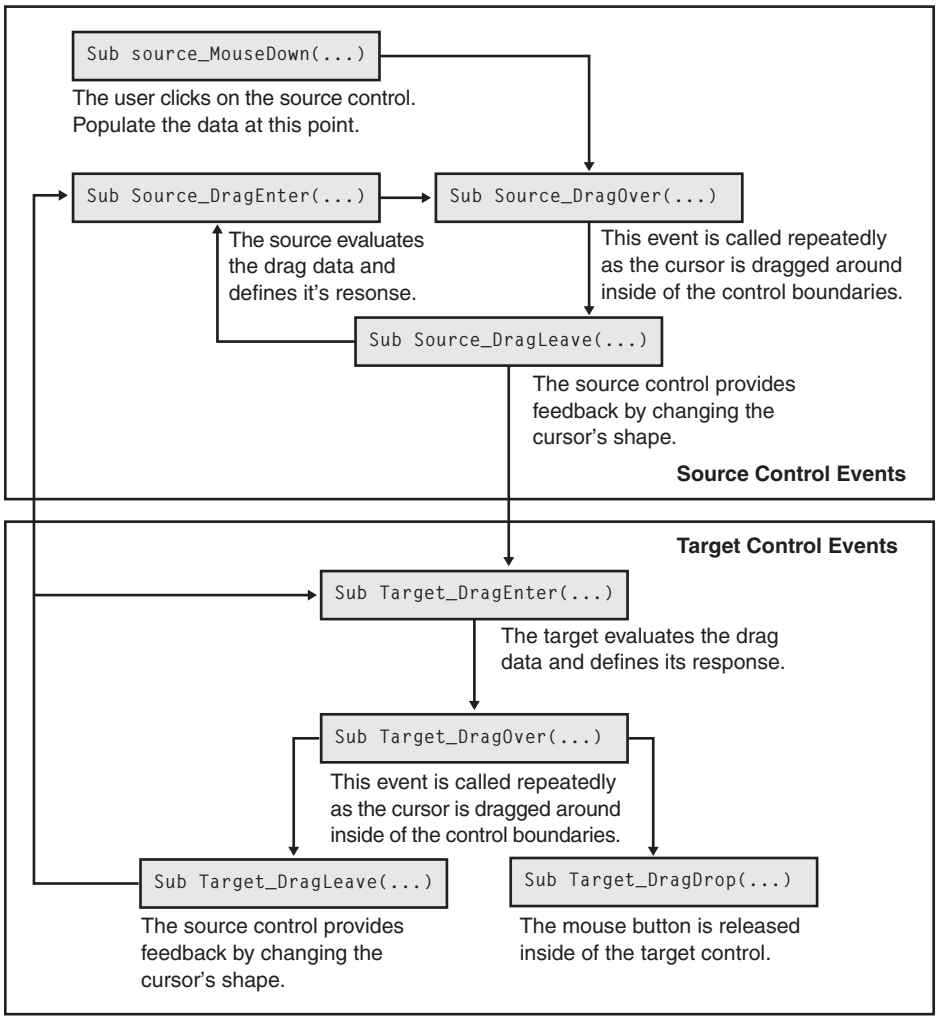


Figure 10-5 Life cycle of a Visual Basic .NET drag-and-drop operation.

The OLEDragAndDrop sample application contains the code necessary to implement the drag-and-drop operation in Visual Basic 6. The following is an excerpt from the Form1.frm file in that project:

```
Private Sub PictureDisplay_MouseDown(Button As Integer, _
   Shift As Integer, X As Single, Y As Single)
   PictureDisplay.OLEDrag
End Sub

Private Sub PictureDisplay_OLEDragDrop(Data As DataObject, _
   Effect As Long, Button As Integer, Shift As Integer, X As Single, _
   Y As Single)
```

```
   If Data.GetFormat(vbCFFiles) Then
      Dim i As Integer
      For i = 1 To Data.Files.Count
         Dim file As String
         file = Data.Files(i)

         If Not EntryExists(file) Then
            Select Case UCase(Right(file, 4))
               Case ".tif", ".jpg", ".gif"
                  FileList.AddItem file
                  FileList.ListIndex = FileList.ListCount - 1

                  Set PictureDisplay.Picture = LoadPicture(file)
            End Select
         Else
            SelectListItem file
         End If
      Next
   End If
End Sub

Private Sub PictureDisplay_OLEStartDrag(Data As DataObject, _
   AllowedEffects As Long)
   Data.Files.Clear

   Data.Files.Add FileList
   Data.SetData , vbCFFiles
   AllowedEffects = vbDropEffectCopy
End Sub
```

The changes necessary to make the drag-and-drop code work in Visual Basic .NET are fairly significant. Due to the changes in the drag-and-drop programming model, the Upgrade Wizard cannot automatically make the modifications for you. It simply leaves the methods alone, leaving it to you to implement the logic in the Visual Basic .NET drag-and-drop event model.

Instead of upgrading the Visual Basic 6 code, we have provided a new Visual Basic .NET implementation of the same features. This helps demonstrate how the drag-and-drop event model has changed and how to properly pass information back and forth. The following code shows how to implement the equivalent drag-and-drop operation from the OLEDragAndDrop sample application in Visual Basic .NET. (This code is also included on the companion CD.)

```
Private Sub Form1_Load(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load
   PictureDisplay.AllowDrop = True
End Sub
```

(continued)

```
Private Sub PictureDisplay_DragDrop(ByVal sender As Object, _
   ByVal e As System.Windows.Forms.DragEventArgs) _
   Handles PictureDisplay.DragDrop
   ' Clear out the existing image and ensure that any resources
   ' are released
   If Not PictureDisplay.Image Is Nothing Then
      PictureDisplay.Image.Dispose()
      PictureDisplay.Image = Nothing
   End If

   Dim files() As String = e.Data.GetData(DataFormats.FileDrop, True)

   Dim i As Integer
   For i = 0 To files.Length - 1
      If Not EntryExists(files(i)) Then
         FileList.SelectedIndex = FileList.Items.Add(files(i))
      Else
         SelectListItem(files(i))
      End If
   Next

   ' Set the picture to the last image added
   PictureDisplay.Image = Image.FromFile(FileList.SelectedItem)
End Sub

Private Sub PictureDisplay_DragEnter(ByVal sender As Object, _
   ByVal e As System.Windows.Forms.DragEventArgs) _
   Handles PictureDisplay.DragEnter
   If e.Data.GetDataPresent(DataFormats.FileDrop) Then
      Dim file() As String = e.Data.GetData(DataFormats.FileDrop, True)

      Select Case UCase(Microsoft.VisualBasic.Right(file(0), 4))
         Case ".tif", ".jpg", ".gif"
            e.Effect = DragDropEffects.Copy
         Case Else
            e.Effect = DragDropEffects.None
      End Select
   Else
      e.Effect = DragDropEffects.None
   End If
End Sub

Private Sub PictureDisplay_MouseDown(ByVal sender As Object, _
   ByVal e As System.Windows.Forms.MouseEventArgs) _
   Handles PictureDisplay.MouseDown
   Dim file(0) As String
   file(0) = FileList.SelectedItem
```

```
   Dim d As New DataObject(DataFormats.FileDrop, file)
   PictureDisplay.DoDragDrop(d, DragDropEffects.Copy)
End Sub
```

Use the life-cycle diagram in Figure 10-5 to help gain an understanding of how to integrate drag and drop into your Visual Basic .NET applications. This example should be a good introduction to the issues you will encounter when dealing with drag and drop.

Collection Classes

Upgrading collections built with the Visual Basic Class Builder utility is fairly straightforward. In the typical case, you will need to uncomment only a single line of code to get the collection to work in Visual Basic .NET. Of course, the ease of the upgrade depends a great deal on what modifications were made to the collection and how closely it resembles the originally generated collection class. In most cases, the Upgrade Wizard will insert a simple ToDo comment that requires a minor modification to get your collection to work as expected. The following example is a simple string collection class implemented in Visual Basic 6 using the Class Wizard Utility:

```
'Local variable to hold collection
Private mCol As Collection

Public Function Add(Key As String, Value As String) As String
   'Set the properties passed into the method
   mCol.Add Value, Key
   Add = Value
End Function

Public Property Get Item(vntIndexKey As Variant) As String
   'Used when referencing an element in the collection
   'vntIndexKey contains either the Index or Key to the collection,
   'which is why it is declared as a Variant
   'Syntax: Set foo = x.Item(xyz) or Set foo = x.Item(5)
   Set Item = mCol(vntIndexKey)
End Property

Public Property Get Count() As Long
   'Used when retrieving the number of elements in the
   'collection. Syntax: Debug.Print x.Count
   Count = mCol.Count
End Property

Public Sub Remove(vntIndexKey As Variant)
```

(continued)

```
   'Used when removing an element from the collection
   'vntIndexKey contains either the Index or Key, which is why
   'it is declared as a Variant
   'Syntax: x.Remove(xyz)

   mCol.Remove vntIndexKey
End Sub

Public Property Get NewEnum() As IUnknown
   'This property allows you to enumerate
   'this collection with the For...Each syntax
   Set NewEnum = mCol.[_NewEnum]
End Property

Private Sub Class_Initialize()
   'Creates the collection when this class is created
   Set mCol = New Collection
End Sub

Private Sub Class_Terminate()
   'Destroys collection when this class is terminated
   Set mCol = Nothing
End Sub
```

Essentially, what will happen when you upgrade this code is that the *NewEnum* property on the collection will be commented out and replaced with a *GetEnumerator* method. The following example is the complete Visual Basic .NET version of the previous collection class as created by the Upgrade Wizard. Notice both the *NewEnum* property and the *GetEnumerator* method.

```
Friend Class StringCollection
   Implements System.Collections.IEnumerable
   'Local variable to hold collection
   Private mCol As Collection

   Public Function Add(ByRef Key As String, _
                       ByRef Value As String) As String
      'Set the properties passed into the method
      mCol.Add(Value, Key)
      Add = Value
   End Function

   Default Public ReadOnly Property Item(ByVal vntIndexKey As Object) _
      As String
      Get
         'Used when referencing an element in the collection
         'vntIndexKey contains either the Index or Key to the
         'collection,
```

```
      'which is why it is declared as a Variant
      'Syntax: Set foo = x.Item(xyz) or Set foo = x.Item(5)
      Item = mCol.Item(vntIndexKey)
   End Get
End Property

Public ReadOnly Property Count() As Integer
   Get
      'Used when retrieving the number of elements in the
      'collection. Syntax: Debug.Print x.Count
      Count = mCol.Count()
   End Get
End Property

'UPGRADE_NOTE: NewEnum property was commented out.
'Public ReadOnly Property NewEnum() As stdole.IUnknown
   'Get
      'This property allows you to enumerate
      'this collection with the For...Each syntax
      'NewEnum = mCol._NewEnum
   'End Get
'End Property

Public Function GetEnumerator() As System.Collections.IEnumerator _
   Implements System.Collections.IEnumerable.GetEnumerator
   'UPGRADE_TODO: Uncomment and change the following line to return
   'the collection enumerator.
   'GetEnumerator = mCol.GetEnumerator
End Function

Public Sub Remove(ByRef vntIndexKey As Object)
   'Used when removing an element from the collection
   'vntIndexKey contains either the Index or Key, which is why
   'it is declared as a Variant
   'Syntax: x.Remove(xyz)

   mCol.Remove(vntIndexKey)
End Sub

'UPGRADE_NOTE: Class_Initialize was upgraded to
'Class_Initialize_Renamed
Private Sub Class_Initialize_Renamed()
   'Creates the collection when this class is created
   mCol = New Collection
End Sub

Public Sub New()
   MyBase.New()
   Class_Initialize_Renamed()
End Sub
```

(continued)

```
    'UPGRADE_NOTE: Class_Terminate was upgraded to
    'Class_Terminate_Renamed.
    Private Sub Class_Terminate_Renamed()
        'Destroys collection when this class is terminated
        'UPGRADE_NOTE: Object mCol may not be destroyed until it is
        'garbage collected.
        mCol = Nothing
    End Sub

    Protected Overrides Sub Finalize()
        Class_Terminate_Renamed()
        MyBase.Finalize()
    End Sub
End Class
```

To enable your collection class, you need to uncomment a single line of code in the *GetEnumerator* method (the Visual Basic .NET equivalent of the *NewEnum* property) and make any necessary additional modifications. The vast majority of cases will not require additional modifications, but the Upgrade Wizard does not take any chances. By requiring you to uncomment the line in the *GetEnumerator* method, it forces you to evaluate whether you do in fact need to make any additional changes. Otherwise, just returning the *Enumerator* from the underlying collection class will be perfectly sufficient. To see this sample in action, check out the Visual Basic .NET version of the CollectionSample project on the companion CD.

Conclusion

This chapter focused on ten common problems involving features within your upgraded application. Hopefully, the discussion has given you a better handle on how to resolve similar issues in your own applications. The samples provided on the companion CD should allow you to see how isolated feature areas can be upgraded. The next chapter covers changes in the Visual Basic language and describes how to fix problems arising from those changes.