This chapter describes the procedures you'll use to debug, register, and distribute your component, and the version compatibility features that allow you to enhance your component without breaking existing applications that use it.

In addition, you'll find topics related to distributing components, including Help files, browser strings, and creating versions of your component for use internationally.

Contents

1

For More Information See "Debugging Your Code and Handling Errors," "International Issues," and "Distributing Your Applications."

1

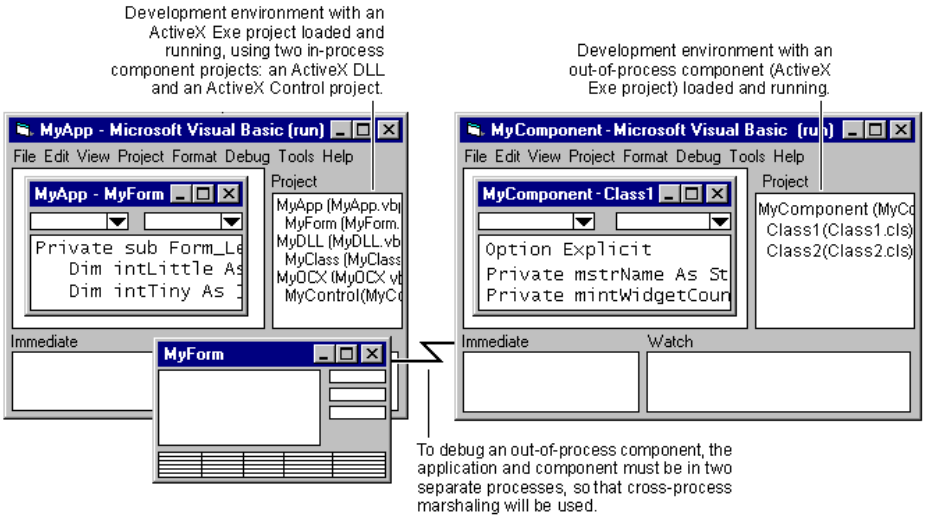
Testing and Debugging ActiveX Components

Visual Basic provides two different component debugging scenarios. For in-process components, you can load a test project (Standard Exe or ActiveX Exe) and one or more component projects into the development environment as a *project group*. You can run all the projects in the group together, and step directly from test project code into in-process component code.

Out-of-process components can be debugged using two instances of the development environment. One instance of Visual Basic runs the test project, while the second runs the component project. You can step directly from test project code into component code, and each instance of Visual Basic has its own set of breakpoints and watches.

Combinations of these scenarios are possible. You can debug an application that uses both in-process and out-of-process components, as shown in Figure 7.1.

Figure 7.1 Debugging in-process and out-of-process components



2

When an ActiveX Exe project is in run mode, like MyComponent in Figure 7.1, the client application (MyApp) can create objects and access their properties and methods. Each out-of-process component a client uses must be in its own instance of the development environment. The client application and all of its in-process components — DLLs and OCXs — can run together in a single instance of the development environment.

For More Information There are special considerations for debugging and testing ActiveX control projects, and other project types that include private controls. See "Setting Up a New Control Project and Test Project" and "Debugging Controls" in "Building ActiveX Controls." The fundamentals of debugging are covered in "Debugging Your Code and Handling Errors."

2

How to Test ActiveX Components

To test a component, you need to create a client application. Components exist to provide objects for clients, which makes it hard to test them by themselves.

Your test project should invoke all the properties, methods, and events of each object provided by your component, testing both valid and invalid values of all arguments.

For example, rather than simply making one call to the Spin method of the Widget object, make a series of calls that try valid and invalid values of all arguments. Pay particular attention to the highest and lowest valid values, as these *boundary conditions* are a frequent source of problems.

Test for both functionality and error cases. Make sure your component behaves well in case of errors, such as unexpected input. It's especially important to make sure

—2

you've covered all error conditions in event procedures of in-process components, because such errors can be fatal to client applications that use the component.

Tip Your test project can also be used to test the compiled component, as described in "How to Test Compiled Components."

3

Make the Test Program Generic for Better Coverage

You can improve your testing process by making the test program more generic. For example, if you create a text box for each argument of the Spin method, and a button to invoke the method, you can use an automated test tool such as Microsoft Test to maintain and run comprehensive test suites. This makes it easier to test combinations of properties and methods.

Testing Components as Part of an Application

If you're creating components as part of an application, you can use the application itself as the test program. In theory, thorough testing of the application will discover any problems with its components.

In practice, however, this is rarely true. An application may not exercise all the interfaces of the components it uses, even under stress testing.

It's also a lot more work to set up test cases when you have to figure out what application behavior must be tested in order to test a particular feature of the component. You'll be better served by a comprehensive test program that directly tests each element of each object's interface.

If each component has been tested separately, testing your application with the components provides an extra level of quality assurance.

Creating a Test Project

The test project must be an Exe project.

The way you set the test project up depends on whether you're testing an in-process or out of process component. The reason for this is explained in "Testing and Debugging ActiveX Components."

For More Information See "Testing and Debugging ActiveX Components" for a list of topics related to testing and debugging.

4

Creating a Test Project for an In-Process Component

This topic describes explains how to set up test projects to exercise most of the objects in-process components can provide.

To create a test project for an in-process component

1 The test project is loaded in the same copy of the development environment where your component project is loaded. On the **File** menu, click **Add Project** to open the **Add Project** dialog box, click the **Standard Exe** icon to select it, then click **OK** to add a Standard Exe project to the project group.

1The caption of the **Project** window changes to **Project Group**, with a default name, to indicate that multiple projects are loaded.

2As described in "How to Test ActiveX Components," use an ActiveX Exe project as your test project if your component implements call-backs.

2 On the **File** menu, click **Save Project Group** to save the group containing the component and test project. From now on, you can open both projects simply by opening the project group.

3 (ActiveX control components skip this step.) Make sure the test project is active — that is, that one of its files is highlighted in the **Project** window. On the **Project** menu, click **References** to open the **References** dialog box. Locate your component in the list, and check it.

1**Note** When setting up a test program for ActiveX control projects, don't set a reference. A control project automatically adds itself to the **Components** dialog box the first time you place a control on a test project form. For additional information, see "Debugging Controls" in "Building ActiveX Controls."

5

3If your component still does not appear in the **References** dialog box of your test project, make sure at least one class module in the component has its Instancing property set to a value *other than* Private and PublicNotCreatable.

4 In the **Project** window, right-click the test project, and click **Set As Start Up** on the context menu to make the test project the one that runs when you press F5.

2**Note** Because ActiveX control projects cannot be startup projects, a test project added to an ActiveX control project will automatically assume the startup role. If the test project entry in the **Project** window is in bold-face type, the test project is already the startup project.

6

5 Add code to test the properties and methods of each public class provided by your component.

3

With the test project selected in the Project window, you can use the Object Browser to verify that the public classes, methods, and properties of your component are available. You can also use the Object Browser to examine and add description strings, and to verify that Help topics are correctly linked.

The view you get in the Object Browser differs depending on which project is currently active — that is, which one is selected in the Project window. When your

component project is active, the Object Browser will show both public members and Friend functions. When the test project is active, only the public members are visible.

For More Information Friend functions are discussed in "Private Communications Between Your Objects" in "General Principles of Component Design." Special considerations for debugging ActiveX control projects, including running code at design time, are covered in "Debugging Controls" in "Building ActiveX Controls."

7

Using Break on Error in Components

You can change the way Visual Basic enters break mode when an error occurs in your component by setting the Error Trapping option in your component project.

In your component project, choose Options from the Tools menu to open the Options dialog box, and select the General tab. There are three options for error trapping, as described below.

Note When you start an instance of Visual Basic, the setting for Error Trapping defaults to Break in Class Module.

8

Suppose you have a component that provides a Widget object that has a Spin method. The following descriptions assume that the test application has called the Spin method of the Widget object, and that an error has occurred in the Spin method's code.

- Break on All Errors: The component project is activated, and the Spin method's code window receives the focus. The line of code that caused the error is highlighted. Visual Basic always enters break mode on such an error, even if error handling is enabled in the Spin method.

  3**Note** You can press ALT+F8 or ALT+F5 to step or run past the error.

9

- Break in Class Module: If error handling is not enabled in the Spin method, or if you are deliberately raising an error for the client by calling the Raise method of the Err object in the Spin method's error handler, the component project is activated, and the Spin method's code window receives the focus. The line of code that caused the error is highlighted.

  4**Note** You can press ALT+F8 or ALT+F5 to step or run past the error.

10

  4If error handling is enabled in Spin, then the error handler is invoked. As long as you don't raise an error in the error handler, Visual Basic does not enter break mode.

- Break on Unhandled Errors: Visual Basic never enters break mode in properties or methods of the component. If error handling is not enabled in the client procedure that called the Spin method, execution stops on the line of code that made the call.

5To understand the behavior of Break on Unhandled Errors in a component project, remember that the component's properties and methods are always called by somebody else. An error in a property or method can always be handled by passing it up the call tree into the client procedure that called the property or method.

4

Note When an out-of-process component enters break mode, focus may not immediately switch to the component project. If you click anywhere on the client, the Component Busy dialog box will be displayed. Click the Switch To button to give the focus to the component project.

11

For More Information See "Debugging Your Code and Handling Errors."

12

How to Test Compiled Components

When you choose Make from the File menu, your component will be registered automatically in the Windows Registry. You can switch your test application between the component project and the compiled component using the procedures in this topic.

In-Process Components

The following procedures perform the switch for an in-process component (ActiveX DLL project or ActiveX control project).

Û To switch from an in-process component project to the compiled .dll or .ocx file

6 On the **File** menu, click **Make <projectname>** to create the compiled in-process component.

7 In the **Project** window, select the component project.

8 On the **File** menu, click **Remove <projectname>** to remove the component project from the project group.

6A warning message will appear: "The project is referenced from another project. Are you sure you want to remove it?" Click **OK** to remove the project.

9 Press F5 to run the test project.

7Visual Basic automatically switches references to the compiled .ocx or .dll file.

5

Ù To switch back to testing your in-process component project

10 On the **File** menu, click **Add Project** to open the **Add Project** dialog box.

11 Use the **Recent** or **Existing** tab to open your component project.

12 Press F5 to run the test project.

8Visual Basic automatically switches references back to the component project.

6

Testing Your Component with Other Applications

You can test your component from any application that can make Automation calls. For example, you can open a Microsoft Excel module, add a reference to your component by choosing References from the Tools menu, and write procedures to create and use objects provided by your component.

Even if you do not expect your component to be used as an extension of end user software tools like Microsoft Excel and Microsoft Access, it's a good idea to test it with such tools. The more programming tools your component works with, the more value it will have for your company or for your customers.

For More Information See "Testing and Debugging ActiveX Components" for a list of topics related to testing and debugging.

13

Generating and Handling Errors in ActiveX Components

There's no such thing as an unhandled error in a component. Untrapped errors in a method of your component, or errors you generate using the Raise method of the Err object, will be raised in the client application that called the method.

Raising errors or returning error codes to the client is the appropriate behavior for components. A well behaved component does not intrude on the client application's user interface by displaying message boxes containing error text.

Users of an application may be blissfully unaware that your component is part of the application they're using. Seeing error messages from a program unknown to them will not improve their day, or help them solve the problem.

For More Information The basics of error handling are discussed in "Debugging Your Code and Handling Errors."

14

Deciding How to Generate Error Messages

When an application calls a method of an object your component provides, there are two general ways in which the method can provide error information.

- Basic-style: The method can raise an error. The client application can implement an error handler to trap errors that may be raised by the method.
- Windows API-style: The value returned by the method can be an error code. The client application can examine the return value to determine whether an error has occurred.

7

There are a number of programming tradeoffs to consider when you select an error generation strategy for your component, but the most important consideration should be the convenience of the developer who will use your component.

For example, if your methods always return an error value, the developer using your component must use *in-line error handling,* that is, the developer must always test the return value after calling a method. This is the way Windows API calls work.

If you raise errors, on the other hand, the developer has the choice of implementing in-line error handling (On Error Resume Next) or writing error-handling routines (On Error GoTo). This flexibility is a hallmark of the coding style familiar to Basic developers.

Be Consistent

Whichever error generation strategy you adopt, be consistent. Developers will not appreciate having to test return values from some methods, and trap errors from others. The more difficult it is to use a component, the less benefit there is from re-using the code.

If you decide to return error values instead of raising them, it's better for all return values to be error codes. This means that if a method also returns a data value, you must use a ByRef parameter for the data. While this is an inconvenience for the user of the method, it's less of an inconvenience than having to test the data type of a return value, to see whether it's an error, before using it as a data value.

Note Client applications that use out-of-process components should always employ some form of error handling, because failures in the underlying cross-process communication layer will be raised as errors in the client application.

15

For More Information "Raising Errors from Your Component" discusses standards and techniques for raising errors from ActiveX components.

16

Guidelines for Raising Errors from Your Component

Use the Raise method of the Err object to raise errors that can be trapped by client applications. When you call the Raise method in the error handler of one of your methods or Property procedures, or when error handling is turned off (On Error GoTo 0), the error will be raised in the client application, in the procedure that directly called your method.

If the procedure that called your method has no error handler, the error condition moves up the call tree of the client until it reaches a procedure that has an error handler, just as any other error would.

When raising an error condition from your component, you don't need to worry about whether the client is written in Visual Basic, Microsoft Visual C++, or another programming language. Any client application can receive the errors your component raises.

Here are a few guidelines you should follow when raising errors from a component.

- The error number you return to client applications is generated by adding an intrinsic constant (vbObjectError) to your internal error number. The resulting value is the one you should document for your users.
- The internal error numbers you add to vbObjectError should be in the range 512 to 65535 (&H200 to &HFFFF). Numbers below 512 may conflict with values reserved for use by the system.
- Establish a “fatal error” or “general failure” code and message for conditions from which your component can’t recover.
- When calling Err.Raise, supply both an error number and a text string describing the error.
- Document your errors in the Help file for your component. For the convenience of your users, you may want to show error numbers in both decimal and hexadecimal format.

8

For example, you might implement the SpinDirection property of the Widget object as a Property procedure, to ensure that it accepts only certain values, as in the following code fragment from the Widget class module of the hypothetical SmallMechanicals component.

```
' Enumeration for SpinDirection property values.
'   (The prefix "sm" identifies it as belonging to the
'   SmallMechanicals component.)
Public Enum smSpinDirection
    smSDClockwise
    smSDCounterClockwise
End Enum

' Module-level storage for SpinDirection property.
Private msdSpinDirection As smSpinDirection

' Implementation of the SpinDirection property.
Property Get SpinDirection() As smSpinDirection
    SpinDirection = msdSpinDirection
End Property

Property Let SpinDirection(ByVal sdNew As _
        smSpinDirection)
    ' The Select Case does nothing if spdNew contains
    ' a valid value.
    Select Case sdNew
        Case smSDClockwise
        Case smSDCounterClockwise
        Case Else
            Err.Raise _
                (ERR_SPN_INVALID + vbObjectError), _
                CMP_SOURCENAME, _
                LoadResString(ERR_SPN_INVALID)
    End Select
```

—9

```
    ' If no error, assign the new property value.
    msdSpinDirection = sdNew
End Property
```

17

The code above assumes that ERR_SPN_INVALID and CMP_SOURCENAME are public constants declared in a standard module, and that error text strings are stored in a resource file.

Because there is no error handler in the Property Let procedure used to set the value of the Spin property, the error is raised in the client application, in the procedure that attempted to set the invalid value.

The text string for the error message is loaded from the component project's resource file, using the internal error number as an index. This technique reduces the amount of memory required to run the component. By concentrating all the text strings in one place, it also simplifies the creation of international versions of the component.

Note Programmers who have used the C++ language will recognize vbObjectError as *facility interface* (FACILITY_ITF), the base constant for the range of errors reserved for a component's interface.

18

For More Information "Handling Errors in a Component" discusses the handling of internal errors, particularly those raised by components your component is using. Providing error messages in multiple languages is discussed in "International Issues."

19

Handling Errors in a Component

When authoring a component, you should be prepared to handle three types of errors:

- Errors generated in your component code that you handle internally.
- Errors generated in your component code that you want to pass back to the client application.
- Errors generated by another component from which your component has obtained object references.

9

Handling Errors Internally

Handling the first of these three error types is no different for a component than for any other application developed using Visual Basic. This type of error handling is covered in "Debugging Your Code and Handling Errors."

Passing Errors Back to the Client

As outlined in "Raising Errors from Your Component," you can use the Raise method of the Err object to raise errors in a client application that calls methods provided by your component. For the error to be raised in the client, you must call Raise from your method's error-handling routine, or with error handling disabled, as in the following fragment of code from the Run method of a hypothetical Widget object.

```
Public Sub Run()
    ' Use in-line error handling.
    On Error Resume Next
    ' (Many lines of Run method code omitted.)
    ' If the following test fails, raise an error in
    ' the client that called the method.
    If intWidgetState <> STATE_RUNNING Then
        ' Disable in-line error handling.
        On Error Goto 0
        Err.Raise _
            Number:=(ERR_WDG_HALTED + vbObjectError), _
            Source:=CMP_SOURCENAME, _
            Description:=LoadResString(ERR_WDG_HALTED)
    End If
    ' (Run method code continues.)
End Sub
```

20

When you raise errors for invalid parameter values, you can simply place the Err.Raise statements at the beginning of the method, before the first On Error statement. The same is true of code to validate property values; place it at the beginning of the Property Let or Property Set.

Raising Errors from Error Handlers

Frequently your properties and methods will include code to handle internal errors. Such code may from time to time contain cases in which no sensible internal response is possible. In such cases, it's reasonable for the property or method to fail and return an error to the client:

```
    On Error Goto ErrHandler
    ' (Code omitted.)
    Exit Sub

ErrHandler:
    Select Case Err.Number
        ' Errors that can be handled (not shown).
        Case 11
            ' Division by zero cannot be handled in any
            ' sensible fashion.
            Err.Raise ERR_INTERNAL + vbObjectError, _
                CMP_SOURCENAME, _
                LoadResString(ERR_INTERNAL)
        ' Other errors (not shown).
    End Select
```

21

In most cases it doesn't make sense to return the original "Divide by zero" error to the client. The client has no way of knowing why such an error occurred within your property or method, so the message is of no use in finding a solution.

If the developer of the client was aware of the possibility of this internal error, because of the excellent documentation you provided, he may have included code to handle it, and shown the end user of the application a useful error message.

If the developer didn't handle it, ERR_INTERNAL might at least serve a diagnostic purpose by including the method parameters in the error message text, as information to pass on to the author of the component.

Handling Errors from Another Component

If your component uses objects provided by another component, you must handle errors that may result when you call methods of those objects. You should be able to obtain a list of these errors from the second component's documentation.

It's important to handle such errors within your component, and not return them to the client application that called your component. The client was written to use your component, and to respond to your component's error codes. The user or programmer who wrote it may have no knowledge of secondary components you're using.

Encapsulation of Errors

The idea that a client application should receive errors only from the components it calls directly reflects the object-oriented concept of *encapsulation*. In the case of errors, encapsulation means that an object is self-contained. Although the object in your component may use other objects supplied by other components, the client application is ignorant of those objects.

For this reason, you should always specify the *source* argument of the Raise method when you handle an error from another component. Otherwise, the Source property of the client application's Err object will contain the name of the component your component called.

Note You should not use the value of the Source property of the Err object for program flow decisions. The Source property may provide useful context information when you're debugging your component, but the text it contains may be version dependent.

22

The following code fragment shows the error handler for a method that uses objects in another component. When it encounters an error it cannot handle, or an unanticipated error, the error handler raises an error for the client, using its own error numbers and descriptions.

```
' Code from the Spin method of a hypothetical Widget
' object.
Public Sub Spin(ByVal Speed As Double)
    On Error Goto ErrHandler
    ' Code containing calls to objects in the Gears
    ' component (omitted).
    Exit Sub

ErrHandler:
    Select Case Err.Number
        ' Other errors (not shown).
        ' --- Errors from the Gears component. ---
        Case vbObjectError + 2000
```

```
            ' Error 2000 is handled by shifting to
            ' another gear (code not shown).
        Case vbObjectError + 3000
            ' Error 3000 causes spin-down; error must be
            ' returned to caller.
            Err.Raise ERR_SPN_SPINDOWN + vbObjectError, _
                CMP_SOURCENAME, _
                LoadResString(ERR_SPN_SPINDOWN)
        ' --- Unanticipated errors from components. ---
        Case vbObjectError To (vbObjectError + 65536)
            Err.Raise ERR_SPN_FAILURE + vbObjectError, _
                CMP_SOURCENAME, _
                LoadResString(ERR_SPN_FAILURE)
        ' Other errors (not shown).
    End Select
End Sub
```

23

Note Some components use older error return mechanisms. Such components may return error numbers in the range 0 – 65535.

24

When the Gears component raises Error 3000, the Spin method must return an error to the application that called it. The error returned is an error of the Spin method. The application using the Spin method doesn't know that Spin is implemented using the Gears component, and could not be expected to deal with Error 3000. All it needs to know is that a spin-down condition has occurred.

In the preceding example a global constant, CMP_SOURCENAME, is used for the *source* argument of the Raise method. If you raise errors outside an error handler, and do not specify the *source* argument, Visual Basic uses the name you entered in the Project Name field of the General tab in the Project Properties dialog box, combined with the Name property of the class module.

While this may be useful for you to know while debugging your component, it's better for the compiled component to specify a consistent value for the *source* argument in all errors it raises.

Note It's frequently easier to use in-line error handling when making calls to components, because the same error may require a different response depending on the method that was called, or the circumstances in which it was called. If you're calling two different components, they may use the same number for different errors.

25

Bending Encapsulation Rules

There may be cases in which some of the error messages returned by a secondary component may contain information of use to the end user of the client application. For example, an error message might contain the name of a file.

In such cases, you may want to bend encapsulation rules to pass along this information. The safest way to do this is to include the text of the Description

property of the Err object in your own error message text, as in the following fragment of error-handling code:

```
' Error 3033 is returned from a call to another
' component; the message text contains a file name that
' will be useful in resolving the problem.
Case vbObjectError + 3033
    ' Raise a Print Run Failure error to the client,
    ' and append the message text from Error 3033.
    Err.Raise _
        ERR_PRINTRUNFAILURE + vbObjectError, _
        CMP_SOURCENAME, _
        LoadResString(ERR_PRINTRUNFAILURE) _
            & Err.Description
```

26

In some cases, it may be useful to the end user to know what component originated the error, and the original error number and message text. You can include all of this information in the description of the error you raise.

For More Information See "Debugging Your Code and Handling Errors."

27

Providing User Assistance for ActiveX Components

Creating a Help file for your component is highly recommended. One of the most important benefits of components is enabling code reuse, a goal you will be much more likely to achieve if developers and end users can easily get Help for the objects you've authored.

You can also provide browser strings that briefly describe your objects and their properties, methods, and events. Users of your component can view these description strings using the Object Browsers in their programming tools, and jump to the Help topics you've provided.

For More Information Creating Help files is discussed in the *Microsoft Windows Help Authoring Kit*, available from Microsoft Press.

When it comes to user assistance, don't underestimate the importance of choosing good names for your objects and their interfaces. See "What's in a Name?" in "General Principles of Component Design," and "Object Naming Guidelines" in "ActiveX Component Standards and Guidelines."

28

How to Specify a Help File for Your Component

▯ To specify a Help file for your component

13 On the **Project** menu, click **Project Properties** to open the **Project Properties** dialog box, then click the **General** tab.

14 In the **Help File Name** box, enter the path and name of the Help file you've created for your component.

15 In the **Project Help Context ID** box, enter the context ID for the specific Help topic to be called when the user clicks the "?" button while your component's type library is selected in the **Object Browser**.

10

For More Information "Providing Help and Browser Strings for Objects" gives the procedure for linking objects and their interface members to Help topics. User assistance features available for components are listed in "Providing User Assistance for ActiveX Components."

29

Providing Help and Browser Strings for Objects

Objects are created from classes. The Object Browser displays information about the classes from which objects are created.

The following procedure can be used to link Help topics and provide browser strings for classes, and for their properties, methods, and events. When Visual Basic creates the type library for your component, it includes this information. Users of your component can view the description strings using the Object Browsers in their programming tools, and jump to your Help topics.

For an alternate method of supplying Help and browser strings for properties, methods, and events, see "Providing Help and Browser Strings for Properties, Methods, and Events."

To enter description strings and link your classes and their members to Help topics

16 Press F2 to open the **Object Browser**. In the **Project/Library** box, select your project.

9If you're not sure which is the **Project/Library** box, hover the mouse pointer over the boxes until you see the tool tip.

17 In the **Classes** list, right click the name of a class to bring up the context menu, and click **Properties** to open the **Member Options** dialog box.

10Alternatively, in the **Members** list, right click the name of a property, method, or event to bring up the context menu, and click **Properties** to open the **Member Options** dialog box.

18 In the **Help Context ID** box, type the context ID of the Help topic to be shown if the user clicks the "?" button when this class or member is selected in the **Object Browser**.

11The path and name of the Help file for the project should appear in the **Help File** box. If it does not, see "How to Specify a Help File for Your Component" for instructions on setting it.

19 In the **Description** box, type a brief description of the class or member.

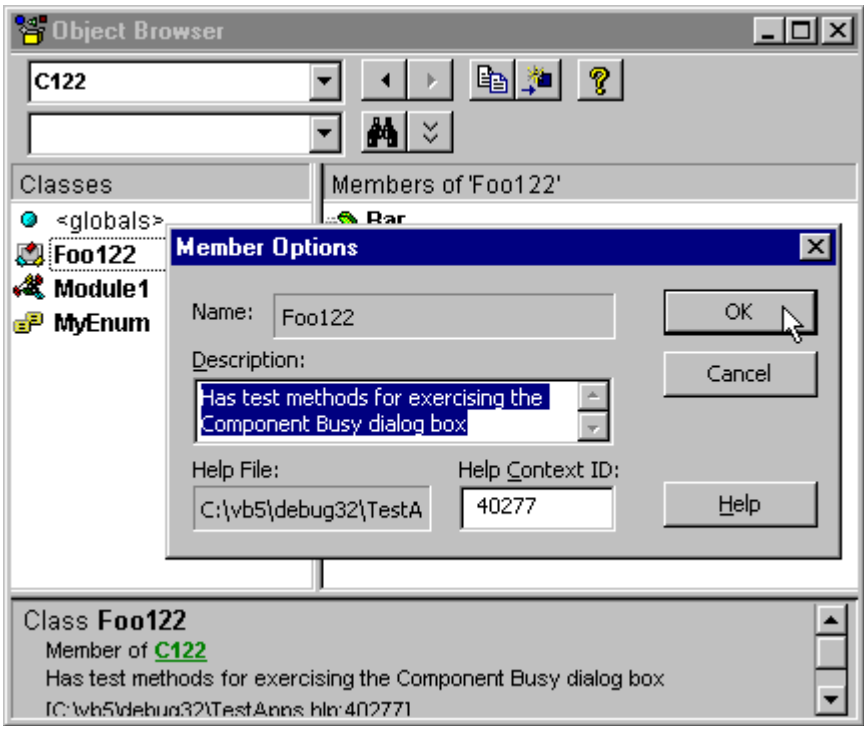
20 Click **OK** to return to the **Object Browser**. The description string you entered should appear on the panel at the bottom of the browser.

21 Repeat steps 2 through 5 for each class and for each member of each class.

11

Figure 7.2 shows the Object Browser and Member Options dialog box as they would appear when setting the description and Help context ID for a class in a hypothetical test application.

Figure 7.2 Setting the description and Help context ID



12

Note You can enter Help context IDs and descriptions for private classes in your component, but this information will only be available to you when you're actually working on your component project. It will not appear in the type library for your component.

You cannot supply browser strings or Help topics for enumerations.

30

For More Information Creating Help files is discussed in the *Microsoft Windows Help Authoring Kit*, available from Microsoft Press. User assistance features available for components are listed in "Providing User Assistance for ActiveX Components."

31

Providing Help and Browser Strings for Properties, Methods, and Events

You can use the Procedure Attributes dialog box to enter description strings for your properties, methods, and events, and to provide links to topics in your Help file. When Visual Basic creates the type library for your component, it includes this information. Users of your component can view the description strings using the Object Browsers in their programming tools, and jump to the Help topics.

You can use the Visual Basic Object Browser to enter description strings for your classes, as described in "Providing Help and Browser Strings for Objects." The Object Browser can also be used to enter this information for properties, methods, and events.

To enter description strings and link Help topics to your properties, methods, and events

22 In the **Project** window, select a module and press F7 (or click **View Code** on the **Project** window toolbar) to open its code window.

23 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box.

24 Select a property, method, or event in the **Name** box.

25 In the **Help Context ID** box, type the context ID of the Help topic to be shown if the user clicks the "?" button when this member is selected in the **Object Browser**.

12The path and name of the Help file for the project should appear in the **Help File** box. If it does not, see "How to Specify a Help File for Your Component" for instructions on setting it.

26 In the **Description** box, type a brief description of the member.

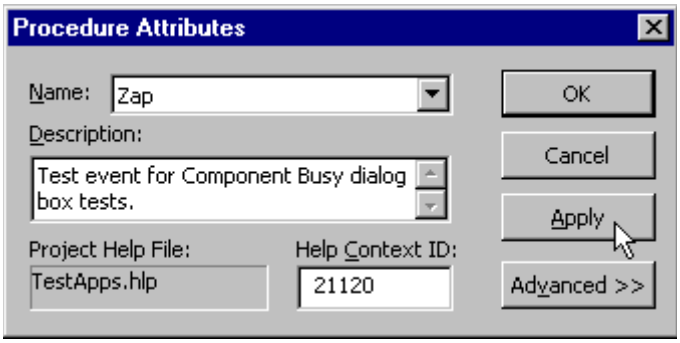
27 Click the **Apply** button to save the information.

28 Repeat steps 3 through 6 for each property and method in the module.

13

Figure 7.3 shows the Procedure Attributes dialog box as it would appear when setting the description and Help context ID for a member of a class in a hypothetical test application.

Figure 7.3 Setting the description and Help context ID



14

Note You can enter Help context IDs and descriptions for members of private classes in your component, but this information will only be available to you when you're actually working on your component project. It will not appear in the type library for your component. You cannot supply browser strings or Help topics for enumerations.

32

For More Information Creating Help files is discussed in the *Microsoft Windows Help Authoring Kit*, available from Microsoft Press. User assistance features available for components are listed in "Providing User Assistance for ActiveX Components."

33

Deploying ActiveX Components

In addition to providing objects for use by client applications, some components can function as standalone desktop applications, in the way Microsoft Excel does. If your component is in this category, you can distribute it as you would any Visual Basic application.

"Distributing Your Applications" contains all the information you need to use the Setup Wizard, or to create a custom Setup for your application.

Ways to Distribute Components

There are several ways to distribute a component. For example:

- As part of your own Visual Basic applications.
- As a tool users can access from Automation-enabled desktop applications such as Microsoft Excel and Microsoft Access.
- As a component other developers can include in their applications, or use with the Internet.
- As part of an enterprise application, running on a remote computer (requires the Enterprise Edition of Visual Basic).

15

For all of these distribution scenarios except the first, you can create a standalone Setup for your component.

Distributing a Component as Part of a Visual Basic Application

To distribute your component as part of a Visual Basic application, you can use Setup Wizard to create a setup program for the application. If your application has a reference to the component, Setup Wizard will locate the component using its registry entries, and include it — along with its support files — in the list of files needed to create distribution media.

As with components that are also standalone desktop applications, this scenario is largely covered by ordinary application setup. The only additional consideration is the use of implemented interfaces.

Including Type Libraries for Implemented Interfaces

If you've used the Implements keyword to add additional interfaces to your classes, as described in "Providing Polymorphism by Implementing Interfaces" in "General Principles of Component Design," you may need to include the type libraries for those interfaces in Setup for your component.

You need to include the type library that includes a particular interface if:

- Any classes the interface has been added to are provided by out-of-process components.
- The application will provide objects to other applications, and some of those objects implement the interface.

16

The reason type libraries need to be included with your application in these two situations is that invoking an object's properties and methods cross-process requires marshaling their arguments. In order to marshal the arguments, type library information must be available.

Standalone Setup for a Component

To distribute your component for use by other developers, by Internet providers, or as part of an Enterprise application, use Setup Wizard to create a standalone setup program. Setup Wizard will automatically include necessary support files. Be sure to include your Help file.

Important For in-process components, see the related topic "Setting Base Addresses for In-Process Components," which contains important information regarding base addresses and their effect on the performance of your component.

34

Developers who use your component can install it on their computers, and then use the Setup Wizard or the Setup Toolkit to include it in the distribution media for their applications.

The steps required to produce Setup for your component will also give you the file dependency information you need to provide to developers who want to use your component with Microsoft Excel, Microsoft Visual C++, or other Automation-enabled development tools.

Distributing Type Libraries for Implemented Interfaces

If you've used the Implements keyword to add additional interfaces to your classes, as described in "Providing Polymorphism by Implementing Interfaces" in "General Principles of Component Design," you need to include the type libraries for those interfaces in Setup for your component.

Special Considerations

Distribution issues particular to ActiveX controls — such as licensing —can be found in "Building ActiveX Controls."

If you plan to use your component for Internet development, Setup Wizard can create CAB files for you. You can obtain the most up-to-date information on Internet setup options from the Microsoft Visual Basic Web site. On the Visual Basic Help menu, click Microsoft on the Web, then click Product News.

Remote Automation support is discussed in the *Guide to Building Client/Server Applications with Visual Basic,* included with the Enterprise Edition of Visual Basic.

For More Information See "Distributing Your Applications."

35

Setting Base Addresses for In-Process Components

In 32-bit operating systems, the code pages for an in-process component (.dll or .ocx file) are shared between processes that use the component, as long as the component can load at its *base address*. Thus three clients could be using the controls in your component, but the code would be loaded into memory only once.

By contrast, if the memory locations used by an in-process component conflict with memory locations used by other in-process components or by the executable, the component must be *rebased* to another logical memory location in the executable's process space.

Rebasing requires the operating system to dynamically recalculate the logical memory locations where code and data are loaded. This recalculation slows down the load process, and code that is dynamically relocated generally *cannot be shared between executables*.

You can greatly improve your component's memory use by choosing a good base address.

Setting the Base Address

To enter the base address for your component, open the Project Properties dialog box and select the Compile tab. The address is entered in the DLL Base Address box, as an unsigned decimal or hexadecimal integer.

The default value is &H11000000 (285,212,672). If you neglect to change this value, your component will conflict with every other in-process component compiled using the default. Staying well away from this address is recommended.

Choosing a Base Address

Choose a base address between 16 megabytes (16,777,216 or &H1000000) and two gigabytes (2,147,483,648 or &H80000000).

The base address must be a multiple of 64K. The memory used by your component begins at the initial base address and is the size of the compiled file, rounded up to the next multiple of 64K.

Your program cannot extend above two gigabytes, so the maximum base address is actually two gigabytes minus the memory used by your component.

Note Executables will usually load at the 4 megabyte logical address. The region below 4 megabytes is reserved under Windows 95, and regions above two gigabytes are reserved by both Windows 95 and Windows NT.

36

Use a Good Random Number Generator

Because there is no way to know what base addresses might be chosen by other in-process components your users might employ, the best practice is to choose an address at random from the indicated range, and round it up to the next multiple of 64K.

If your company produces many in-process components, you may wish to randomly calculate the base address of the first, and then arrange the others above or below the first, thus guaranteeing at least that your company's components will not have memory conflicts.

Version Compatibility in ActiveX Components

A component can be part of another application because it provides Automation interfaces that the other application can manipulate. Each public class module has a default interface that includes all the properties and methods you added to the class module, plus any secondary interfaces implemented using the Implements feature.

Once your component has been used in an application — or, in the case of ActiveX controls, embedded in a document or on a Web page — you can change its interfaces only at the risk of breaking the client application.

Suppose, for example, that the Spin method of your Widget object has one argument, Speed. If you distribute a new version of your component, in which you redefine the Spin method so that it also requires a Direction argument, you could cause run-time errors in existing applications.

At the same time, a successful component will inevitably spark requests for enhancements. You will want to provide new objects, or add new properties and methods to existing objects. Occasionally you will even want to change the arguments of existing methods of existing objects.

For More Information See "Polymorphism, Interfaces, Type Libraries, and GUIDs" in "General Principles of Component Design" for background information and concepts.

37

When Should I Use Version Compatibility?

Visual Basic provides two mechanisms for maintaining backward compatibility while enhancing software components — the Version Compatibility feature and the Implements feature.

Version Compatibility

Visual Basic's Version Compatibility feature is a way of enhancing your components while maintaining backward compatibility with programs that were compiled using earlier versions. The Version Compatibility box, located on the Component tab of the Project Options dialog box, contains three options:

- No Compatibility: Each time you compile the component, new type library information is generated, including new class IDs and new interface IDs. There is no relation between versions of a component, and programs compiled to use one version cannot use subsequent versions.
- Project Compatibility: Each time you compile the component, new type library information is generated — but the type library identifier is kept, so that your test projects can maintain their references to the component project.

  1**Important** For the purpose of releasing compatible versions of a component, Project Compatibility is the same as No Compatibility.

38

- Binary Compatibility: When you compile the component, Visual Basic creates new class IDs and interface IDs only if necessary — and preserves the class ID and interface ID information from the previous version, so that programs compiled with the earlier version will continue to work.

  13Visual Basic also warns you when changes to your code would make the new version incompatible with previously compiled applications.

  5**Note** When people talk about Version Compatibility, they're usually referring to Binary Compatibility.

17

The appropriate use of these options is described below.

Using the Implements Statement for Compatibility

The Implements statement allows you to add multiple interfaces to class modules, as described in “Polymorphism, Interfaces, Type Libraries, and GUIDs” and “Providing Polymorphism by Implementing Interfaces” in “General Principles of Component Design,” and in “Polymorphism” in “Programming with Objects.”

Multiple interfaces allow your systems to evolve over time, without breaking existing components or requiring massive re-compiles, because a released interface is never changed. Instead, new functionality is added to a system by creating new interfaces.

This approach is much more in keeping with the design philosophy of the Component Object Model (COM), on which the ActiveX specification is based.

Note The Binary Compatibility option of Version Compatibility is useful in conjunction with Implements and multiple interfaces, to prevent changes to the default interfaces of your classes.

39

When to Use Version Compatibility Options

If you decide to use the Version Compatibility feature, you may find the following rules helpful in determining when to use the different options:

Use No Compatibility to Make a Clean Break

When you begin working on a new version of an existing component, you may decide that the only way to make necessary enhancements is to break backward compatibility. In this case, set No Compatibility the first time you compile your project. This guarantees that you’ll start with a clean slate of identifiers, and that existing programs won’t mistakenly try to use the incompatible version.

You should also change the file name of your component at this time, so that the incompatible version won’t over-write earlier versions on your users’ hard disks.

After compiling once with No Compatibility, switch to Project Compatibility to simplify your development tasks.

Use Project Compatibility for New Development

Use the Project Compatibility setting when you’re developing the first version of a component. Project Compatibility preserves the type library identifier, so that you’re not continually setting references from your test projects to your component projects.

Using Project Compatibility also makes it easier to switch between the component project and the compiled component when you’re testing.

Project Compatibility is discussed in “Project Compatibility: Avoiding MISSING References.”

Use Binary Compatibility for New Versions of Existing Components

Switch to Binary Compatibility mode when you begin work on the second version of any component, if you want applications compiled using the earlier version to continue to work using the new version.

Switching to Binary Compatibility is discussed in the related topic "Providing a Reference Point for Compatibility."

Don't Mix Binary Compatibility and Multiple Interfaces

If you use multiple interfaces and the Implements statement to provide backward compatibility, don't use Binary Compatibility to modify the abstract interfaces you've defined for use with Implements.

If you enhance any of the interfaces in a component, Visual Basic will change the interface IDs. The technique of evolving component software by adding interfaces depends on *interface invariance.* That is, an interface once defined is never changed — including the interface ID.

For More Information See "Providing Polymorphism by Implementing Interfaces" in "General Principles of Component Design" for information about component software design using multiple interfaces. "Maintaining Binary Compatibility" describes the versioning system Visual Basic uses to prevent compatibility problems.

40

Maintaining Binary Compatibility

Visual Basic maintains backward compatibility by preserving class ID and interface ID information from previous versions of your component, as described in "Polymorphism, Interfaces, Type Libraries, and GUIDs," in "General Principles of Component Design." This information is not maintained in the type library, but is stored elsewhere in your component.

When a client application is compiled using a particular version of your component, the class ID and interface ID of each object it uses will be compiled in. When the client is run, the class ID is used to create an instance of the class, and the interface ID is used to verify that it's safe to make the property and method calls that were compiled into the client.

If you make a new version of your component, using the Binary Compatibility option, the new version will contain the class IDs and interface IDs the old client needs to create objects and use their properties and methods, as well as class IDs and interface IDs of the enhanced versions of the classes.

New client applications compiled using the new version of your component can make use of the enhancements, because they compile in the new class IDs and interface IDs.

For example, suppose that in version 1.0 of your component, your Widget object has a Spin method, and that you've compiled a client application that creates a Widget object and calls the Spin method.

Now suppose you set the Binary Compatibility option and compile a new version of your component, in which the Widget object also has an Oscillate method. Visual Basic creates a new class ID and interface ID for the enhanced Widget. Because adding a method doesn't break binary compatibility, Visual Basic also maintains the class ID and interface ID of the old Widget.

When your previously compiled application uses these old IDs, it gets an enhanced Widget — which is fine, because the new Widget still has the Spin method the old application needs to call.

Note Binary Compatibility applies only to the default interface of a class — that is, the Public Sub, Function, and Property procedures you add to the class module. Interfaces you add using Implements are ignored.

41

Incompatible Interface Changes

Suppose that instead of adding an Oscillate method, you changed the arguments of the Spin method. For example, you might add a Direction argument.

If you could compile your component using the old class ID and interface ID for the Widget class, your old client application would be in trouble. It would be able to create the new Widget, but when it called the Spin method it would put the wrong arguments on the stack. At the very least, a program error would occur. Even worse, data could be corrupted.

Preventing Incompatibility

If you've selected the Binary Compatibility option, Visual Basic warns you when you're about to compile an incompatible version of your component. You can reverse the edits that would make your component incompatible, or change the file name and Project Name so that the new version will not replace the old when users run Setup.

If you choose to disregard the warnings, and compile an incompatible version of your component with the same file name and Project Name, Visual Basic dumps all of the class IDs and interface IDs from previous versions of your component.

When the incompatible component is installed on a computer that has a client application compiled using an earlier version, it will overwrite the earlier version. Subsequently, when the client application attempts to create objects, it will receive error 429, "OLE Automation server cannot create object."

This averts more serious and subtle errors that might occur when the application attempts to invoke the properties and methods of the incompatible interface.

Limited Protection For Late-Bound Client Applications

Late binding is used when variables are declared As Object, because the compiler doesn't know the class ID of the objects and interfaces that may be assigned to the variable at run time. Applications that use late binding create instances of your classes using the CreateObject function and the programmatic ID, as shown here:

```
Dim obj As Object
Set obj = CreateObject("MyComponent.MyObject")
```

42

The CreateObject function looks up the class ID in the Windows Registry, and uses it to create the object. Thus it will always create the most recent version of the object.

As long as you preserve binary compatibility, late-bound clients will continue to work successfully with your component.

If you make an incompatible version of your component using the same programmatic IDs for your objects, late-bound clients can still create the objects, because they're looking up the class ID instead of having it compiled in. When they call methods whose arguments have changed, or methods you've deleted, program failure or data corruption may occur.

For More Information See "Levels of Binary Version Compatibility" for a description of the degrees of compatibility Visual Basic measures. See "Version Compatibility" for a list of topics related to this feature. See "Polymorphism, Interfaces, Type Libraries, and GUIDs," in "General Principles of Component Design" for background information and concepts.

43

Levels of Binary Version Compatibility

Visual Basic defines three levels of version compatibility for the interfaces you describe in your class modules.

- *Version identical* means that the interfaces are all the same, so the new version of the type library is exactly the same as the old one. The code inside methods or Property procedures may have been changed or enhanced, but this is transparent to client applications.
- *Version compatible* means that objects and/or methods have been added to the type library, but no changes were made to existing properties or methods. Both old and new client applications can use the component.
- *Version incompatible* means that at least one property or method that existed in the old type library has been changed or removed. Existing client applications that have references to the component cannot use the new version.

18

Version-Identical Interfaces

Once your component has been distributed as part of an application, there are several situations that might cause you to release an update. You might want to optimize the performance of a method that had turned out to be a bottleneck for users. You might

also need to change the internal implementation of an object's method to reflect changes in the business rule on which the method was based.

You can change the code in existing Property procedures or methods, and still have a version-identical interface, as long as you do not change the names or data types of their parameters, the order of the parameters, the name of the property or method, or the data type of the return value.

When you create the executable for a version-identical upgrade, you can use the same file name for the executable. Visual Basic uses the same version number for the type library.

Important When you release a new version of your component with a version-identical or version-compatible interface, and retain the same file name for the executable, you should always use the Make tab of the Project Properties dialog box to increment the file version number. This ensures that the setup programs for applications that use your component will replace old versions during setup.

44

Version-Compatible Interfaces

When you enhance your component by adding new classes, or new properties and methods to existing classes, you can continue to use the same name for your executable. As long as you make no changes to existing properties and methods, Visual Basic updates the version number of the type library but keeps it compatible with the old version number.

Client applications that are built using the new version of your component will compile with the new version number, and can make use of all the new features. They cannot be used with earlier versions of your component, however, because type library versions are only upward-compatible.

As with version-identical releases, remember to increment the file version number of the executable.

Version-Incompatible Interfaces

Sometimes design decisions made in an earlier version of a component fail to anticipate future needs. If you want the code in the component to be useful in new development projects, you have to change the interface.

For example, the CupsPerAnnum parameter of the Coffee method might be implemented as an Integer in the first version of a component. It may become apparent, after the component has been in use for some time, that some clients need to pass a larger value than can be contained in an Integer.

Changing the declaration of a method is only one of several actions that will cause Visual Basic to make the version number of the type library incompatible, rendering

the new version unusable with client applications compiled with earlier versions. The following changes will cause a version incompatibility:

- Changing the Project Name field on the General tab of the Project Options dialog box.
- Changing the Name property of any class module whose Public property is True (controls), or whose Instancing property is not Private (class modules).
- Deleting a public class module, or setting its Instancing property to Private.
- Deleting a public variable, procedure, or Property procedure from a public class module or control, or changing it to Private or Friend.
- Changing the name or data type of a public variable, procedure, or Property procedure in a public class module or control.
- Changing the names, data types, or order of the parameters of a public procedure or Property procedure in a public class module or control.
- Changing the Procedure ID (DispID) or other parameters in the Procedure Attributes dialog box.

19

Time to Take Stock

When you've identified a necessary change that will cause your component to be incompatible with earlier versions, it's a good idea to take the time to evaluate the entire set of interfaces, before plunging ahead and creating an incompatible version of your component.

Consider Multiple Interfaces

Remember that there are alternatives to using Version Compatibility. Consider enhancing your component by adding new interfaces with the Implements statement, as described in "Providing Polymorphism by Implementing Interfaces" in "General Principles of Component Design."

Multiple interfaces, a key feature of the Component Object Model (COM) — on which the ActiveX specification is based — provide a more flexible way to enhance software components. They allow you to evolve your systems over time, without breaking existing components.

You don't have to tackle the daunting task factoring your existing class module interfaces into small interfaces more suitable for use with Implements — one of the benefits of using multiple interfaces is that you can start small, adding new interfaces to the system only where new functionality is required.

Going Ahead with Incompatibility

If you decide to go ahead with an incompatible version, you can minimize future problems for the users of your component by concentrating in one release all the changes you can anticipate that might break compatibility again if they have to be made in later releases.

—28

In planning for an incompatible change, treat the project as a fresh start. Devote as much care to planning as you would if you were creating a brand new component.

Changing the Project Name

The key change you must make, when you need to distribute an incompatible version of your component, is the project name. The project name, which is set on the General tab of the Project Properties dialog box, is the first part of the programmatic ID of each class your component provides.

For example, the SmallMechanicals component might provide a Widgets class. A client application would create a variable to contain a reference to a Widget object as follows:

```
Private wdgDriver As SmallMechanicals.Widget
```

45

The programmatic ID is the combination of project name and class name, and it must be unique. If you create a new version of this component, you might give it the project name SmallMechanicals200. Both versions of the Widget object could then be registered in the same Windows Registry without confusion.

Note You must also change the file name of an incompatible component.

46

Alternatives to Version-Incompatible Changes

If you prefer not to make the change to multiple interfaces, as described above, you can take a similar approach with classes.

That is, you can avoid changes that cause version incompatibility by adding new objects, properties, and methods, instead of changing existing ones. Existing applications continue using the old methods and objects, while developers of new applications can use the new objects.

For example, you might discover that to take advantage of enhancements to your General Ledger system, you need to add a SubAccount parameter to several business rules in your FinanceRules component.

If each rule is implemented as a method of the GL object, you could avoid creating an incompatible version of the component by adding a new object named GL97. This object would have the same methods as the GL object, but with a SubAccount parameter where appropriate.

If you need to add new versions of existing methods to an object, you can give the new methods the same name with a version or sequence number added — for example, ‘Execute2.’

This approach is not as flexible or efficient as implementing multiple interfaces. You may end up replicating entire classes, and class interfaces may become large and unwieldy — for example, you might find yourself using a Query class with an Execute method, an Execute2 method, an Execute3 method, and so on. However, it’s a step in the right direction.

For More Information "Providing a Reference Point for Binary Version Compatibility" describes when and how to specify a version of your component as a reference point for version compatibility. See "Version Compatibility" for a list of topics related to this feature. Software evolution using multiple interfaces is discussed in "Providing Polymorphism by Implementing Interfaces," in "General Principles of Component Design."

47

Providing a Reference Point for Binary Version Compatibility

To determine the degree of compatibility between two versions of a component, Visual Basic needs a reference point. You provide this reference point by entering the path to a previously compiled version of your component in the Version Compatibility box on the tab of the Project Properties dialog box.

You need to do this whenever you begin work on a new version of a component you have shipped, put into production, or used as part of an application.

To specify a reference version of the component type library

29 Open the project.

30 From the **Tools** menu, choose **Project Properties** to open the **Project Properties** dialog box, and select the **Component** tab.

31 Click **Binary Compatibility** to lock down the class IDs in the project.

6**Note** As explained in "Project Compatibility: Avoiding MISSING References," the **Project Compatibility** setting actually has nothing to do with the Version Compatibility feature.

48

32 Update the box at the bottom of the **Version Compatibility** frame with the full path and name of the most recent version of your component.

20

Whenever you make a new executable from your component project, Visual Basic compares the new interfaces of your classes to the ones described in the file you have specified. Visual Basic updates the type library version number according to the level of compatibility between the interfaces.

For More Information "Using Binary Version Compatibility" describes when and how to use the feature, problems you may encounter, and messages you may get from Visual Basic. See "Version Compatibility" for a list of topics related to this feature.

49

Using Binary Version Compatibility

Whenever you begin work on a new version of an existing component, you need to specify a type library which Visual Basic can use as a reference point for determining

compatibility. In most cases, this will be the type library included in executable (.exe, .dll, or .ocx file) for the last version of the component you distributed.

Each time you build an interim version of your updated component, Visual Basic extracts information about the old interfaces from this .exe file and compares it to the new interfaces of your class modules.

"Providing a Reference Point for Compatibility" explains the procedure for creating a reference point.

Keeping the Reference Version Separate from Interim Builds

Important Keep the copy of the .exe file you specify as your reference version separate from the build copy of the new version.

50

Each time you make an interim build, Visual Basic adds a new set of interface identifiers to the executable, one identifier for each class module. If you specify your build copy as the reference version, it will accumulate a complete set of interface identifiers for every version-compatible interim build you have ever done. (Interface identifiers do not change for version-identical builds.)

In addition to the sixteen bytes taken up by each interface identifier, having unused interface identifiers in your executable — only your test applications ever use the interim versions — can slow down cross-process access to your component in some situations, and the Windows Registry of any computer your component is installed on will be cluttered with unused interface identifiers.

If your reference version is a copy of your last released executable, all your interim builds will have the same interface version number, and your final build will have only the interface identifiers it needs: all the sets from the reference version (to provide backward compatibility) plus the set of interface identifiers for all the classes in your new release.

Note When you're developing the first version of a component, using Project Compatibility instead of Binary Compatibility, exactly the opposite is true: The reference version should be your interim build. This does not bulk up the type library, because Project Compatibility *never* saves the interface identifiers.

51

Avoiding Version Trees

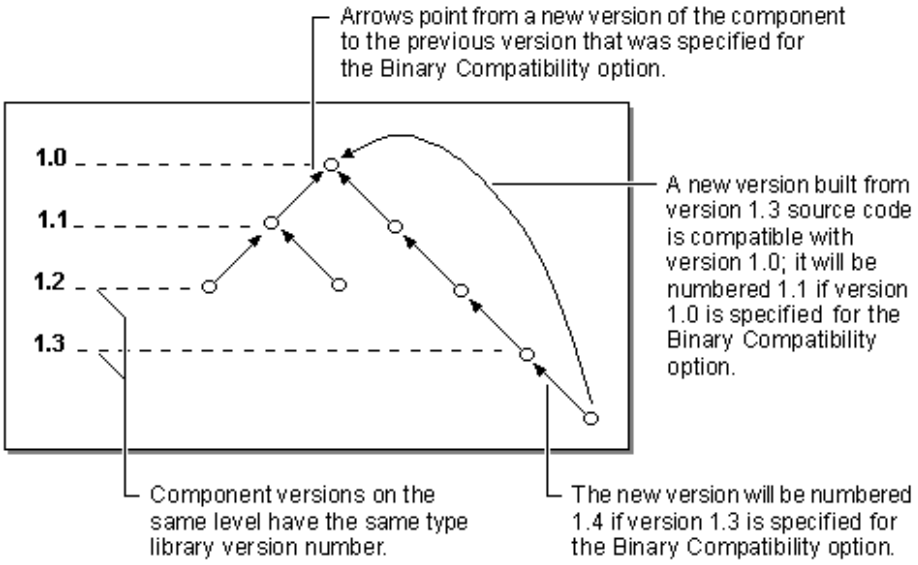
Because Visual Basic produces the version number for the new type library by incrementing the type library version number it finds in the reference version, the released versions of your component form a chain, each link derived from its predecessor. As mentioned earlier, each new release contains all the interface identifiers for preceding versions, so that a client application compiled with any previous version will still work with the latest.

What's a Version Tree?

Version trees arise when your component's version history acquires branches — that is, when you produce two physically distinct components based on the same source code. It's important to keep the version history of your component straight, and avoid such branching.

Figure 7.4 shows some of the problems that can be caused by version trees. (The version numbers are for illustrative purposes only, and are not intended to represent actual type library version numbers.)

Figure 7.4 Problems with version trees



21

The long branch at the right shows four successive versions of a component executable, and a new executable that has been created by adding to the source code for the executable whose type library version is 1.3.

The correct continuation of this version history is for the latest executable to be compiled with the version 1.3 executable as its reference version. The new type library version number is 1.4. The .exe file maintains compatibility with client applications compiled using any of its predecessors.

Because it's at the end of a chain of compatible versions, the new executable could also be compiled with the version 1.0 executable as its reference version. In this case, its type library version number will be 1.1. This could cause problems for clients compiled to take advantage of features of the new version. If they're placed on a computer with the earlier version 1.1 executable, the new features will not be available, and the applications will fail.

A different problem arises when the new component is installed on computers that have client applications compiled with type library version numbers 1.1, 1.2, and 1.3. Standard practice is to increase the file version number of each new executable file, to ensure that Setup will replace earlier versions of the executable. (Remember that file version numbers are independent of type library version numbers.)

Thus the new executable, containing interface identifiers for type library versions 1.0 and 1.1, will replace older executables that contained interface identifiers for type library versions 1.0 through 1.3.

If the computer already has client applications compiled with type library versions 1.2 and 1.3, those clients will be unable to use the new version of the component.

Divergent Versions

The left side of the tree shows divergent versions. This can arise when the source code for an early version of your component is taken as the basis of a new component, and classes are added that do not exist in your main version history.

If the executable for your component is used as the reference version for the divergent version, the type library version numbers of the divergent version and its successors will overlap the version numbers of your components. The results for client applications will be disastrous.

Tip You can easily avoid the creation of version trees by always setting aside a copy of the previous version of your component's executable file (.exe, .dll, or .ocx) as the reference version for the next release, as described earlier in this topic.

52

Tip If you decide to use the source code of an earlier version of your component as the basis of a new component, give the new component a different project name and executable name.

53

Version Trees with Project Compatibility

Version trees can also arise when you're using Project Compatibility, the difference being that it's the major version number of the type library that changes (instead of the minor version number, as shown in Figure 7.4). The consequences to client applications can be equally disastrous.

As with Binary Compatibility, the best way to avoid this is not to split your source tree. If you take a copy of your source code at a particular stage of the project as the basis for another component, use a different project name and executable name for this new project.

Version Compatibility Messages

For performance reasons, Visual Basic does not fully compare interfaces as you edit. When you run your component project, Visual Basic will always display a compatibility warning if the new version is incompatible with the old. (Version-

identical and version-compatible interfaces will compile without compatibility warnings.)

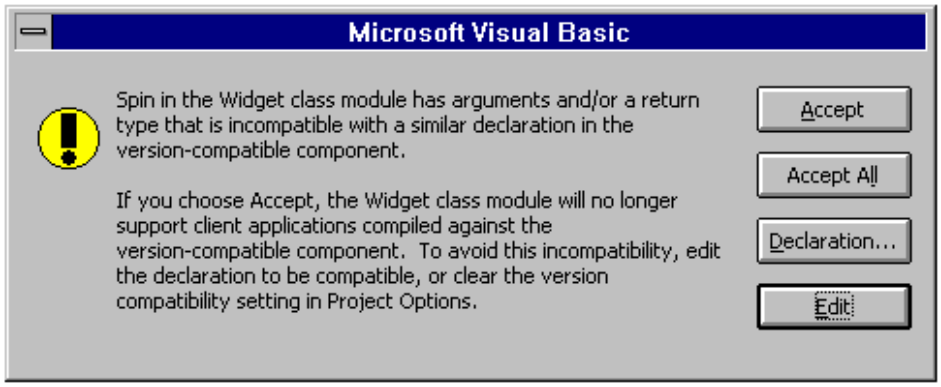
Note Version compatibility is judged on a project-wide basis. A change in the declaration of just one method in one class module causes the entire project to be marked as incompatible with the previous version. See “Levels of Version Compatibility,” earlier in this chapter, for a listing of changes that will cause a version incompatibility.

54

Version Incompatibility Warnings

Suppose you add a new argument to the Spin method of the Widget object. When you run the project, you’ll get a warning like that shown in Figure 7.5.

Figure 7.5 Version incompatibility warning



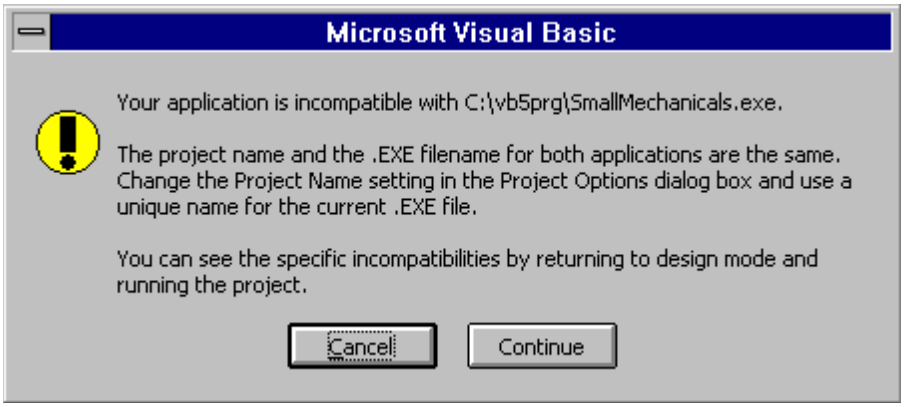
22

You can examine the old declaration by clicking the Declaration button on this message. If you made the change accidentally, you can click Edit to bring up the code and fix it.

If the change was intentional, you can click Accept. The new version of your component is now incompatible with the previous version. If you have many such changes, you can click Accept All to avoid getting a warning for each one.

If you have not changed the project name, when you use the Make EXE File command, you will get a warning that your application is incompatible with the .exe file you specified as your reference version, as shown in Figure 7.6.

Figure 7.6 Warning for incompatible .exe file



23

Clicking Continue at this point creates an executable file that could cause existing client applications to fail. It's highly recommended that you change the file name as well as the project name when you create an incompatible version of a component.

For More Information See "Version Compatibility" for a list of topics related to this feature.

55