This chapter and “Debugging, Testing, and Deploying Components” contain those topics that apply to all types of ActiveX components. These chapters provide necessary background for the in-depth treatment of component types in subsequent chapters.

Note If you are using the Control Creation Edition of Visual Basic, some of the material covered in this document may not be applicable. With the full editions of Visual Basic you have the ability to create applications, ActiveX documents, and other types of components. Although some of the terminology may relate to application specific objects such as forms, in most cases the underlying concepts also apply to ActiveX control creation.

1

This chapter begins with “Component Basics,” a group of topics that explain key terminology and concepts of component design.

The rest of the topics in "General Principles of Component Design" and “Debugging, Testing, and Deploying Components” are organized according to the general sequence of development tasks for components:

1. Determine the features your component will provide.
2. Determine what objects are required to divide the functionality of the component in a logical fashion.

   1See “Adding Classes to Components.”
3. Design any forms your component will display.
4. Design the interface — that is, the properties, methods, and events — for each class provided by your component.

   2See “Adding Properties and Methods to Classes,” “Adding Events to Classes,” “Providing Named Constants for Your Component,” “Providing Polymorphism by Implementing Interfaces,” and “Organizing Objects: The Object Model.”

   3The remainder of the task-oriented topics are contained in “Debugging, Testing, and Deploying Components.” In addition to the following development tasks, they cover distribution, version compatibility, and creating international versions of your component.
5. Create the component project and test project.
6. Implement the forms required by your component.
7. Implement the interface of each class, provide browser strings for interface members, and add links to Help topics.
8. As you add each interface element or feature, add code to your test project to exercise the new functionality.
9. Compile your component and test it with all potential target applications.

1

Contents

Component Basics

A software component created with Visual Basic is a file containing executable code — an .exe, .dll, or .ocx file — that provides objects other applications and components can use.

An application or component that uses objects provided by other software components is referred to as a *client*. A client uses the services of a software component by creating instances of classes the component provides, and calling their properties and methods.

In earlier versions of Visual Basic, you could create components called *OLE servers.* The features of components created with Visual Basic are greatly expanded, including the ability to raise events, improved support for asynchronous call-backs, and the ability to provide ActiveX controls and documents.

In-Process and Out-of-Process Components

An application or component that uses objects provided by another component is called a *client.*

Components are characterized by their location relative to clients. An *out-of-process* component is an .exe file that runs in its own process, with its own thread of execution. Communication between a client and an out-of-process component is therefore called *cross-process* or *out-of-process* communication.

An *in-process* component, such as a .dll or .ocx file, runs in the same process as the client. It provides the fastest way of accessing objects, because property and method calls don’t have to be marshaled across process boundaries. However, an in-process component must use the client’s thread of execution.

What’s in a Name?

The names you select for your class modules and for their properties, methods, and events make up the interface(s) by which your component will be accessed. When naming these elements, and their named parameters, you can help the user of your component by following a few simple rules.

- Use complete words whenever possible, as for example "SpellCheck." Abbreviations can take many forms, and hence can be confusing. If whole words are too long, use complete first syllables.
- Use mixed case for your identifiers, capitalizing each word or syllable, as for example ShortcutMenus, or AsyncReadComplete.
- Use the same word your users would use to describe a concept. For example, use Name rather than Lbl.
- Use the correct plural for collection class names, as for example Worksheets, Forms, or Widgets. If the collection holds objects with a name that ends in "s," append the word Collection, as for example SeriesCollection.
- Use a prefix for the named constants in Enums, as discussed in "Providing Named Constants for Your Component," later in this chapter.
- Use either the verb/object or object/verb order consistently for your method names. That is, use InsertWidget, InsertSprocket, and so on, or always place the object first, as in WidgetInsert and SprocketInsert.

2

Choosing a Project Type and Setting Project Properties

When you open a new project, you have three choices for project type: ActiveX EXE, ActiveX DLL, and ActiveX Control. The type you choose determines what kinds of objects your component can provide.

The following list may assist you in selecting the correct project type for your component.

10. If your component is going to provide ActiveX controls, open a new ActiveX control project. Controls can only be provided by control components (.ocx files), which must be compiled from ActiveX control projects. Control components always run in process.

    1**Note** Control components are limited in their ability to provide other kinds of objects because class modules in ActiveX control projects can only have two Instancing settings, PublicNotCreatable or Private. Objects in control components are best used to enhance the features of controls; put objects with other uses in a separate ActiveX DLL project.

11. If you're creating an out-of-process component, open a new ActiveX EXE project. Reasons to create an out-of-process component include:

- The component can run as a standalone desktop application, like Microsoft Excel or Microsoft Word, in addition to providing objects.

12.If you're creating an in-process component, open a new ActiveX DLL project. Reasons to create an in-process component include:

- An in-process component shares its client's address space, so property and method calls don't have to be marshaled. This results in much faster performance.

3

Once you've opened a project for your new component, there are some project properties you should always set.

To set properties for a new component project

1 On the **Project** menu, click **Project1 Properties** to open the **Project Properties** dialog box.

2 On the **General** tab, set the **Project Name**.

4This is the most important property of any new component. It identifies your component in the Windows registry and the Object Browser; its uniqueness is therefore important.

5It's also the default name of the compiled component, and the name of the *type library* that contains descriptions of the objects and interfaces provided by your component. See "Polymorphism, Interfaces, Type Libraries, and GUIDs," later in this chapter.

3 On the **General** tab, set the **Project Description**.

6Project description is the text string a developer or user will see when setting a reference to your component, or when selecting your control component in the **Components** dialog box.

4 On the **General** tab, set the **Startup Object**.

7Click **None** if there is no code you need to execute to initialize your component. If your component requires initialization, click **Sub Main**, add a module to your project, and in that module declare a Public Sub named Main. Place your initialization code in this Sub procedure.

1**Important** See "Starting and Ending a Component" later in this chapter for an explanation of the need to keep the processing in Sub Main to a minimum.

4

2**Note** Do not place Sub Main in a class module. Placing Sub Main in a class module turns it into a method named Main, rather than a startup procedure.

5

Setting Other Properties

Depending on the type of component you're creating, other project properties may be of interest to you.

Help File Name and Project Help Context ID

Providing a Help file for your component is highly recommended. See “Providing User Assistance for ActiveX Components” in “Debugging, Testing, and Deploying Components” for information on linking help topics to the properties and methods of the classes your component provides.

Make Tab Properties

Properties on the Make tab allow you to control file version numbers and version information about your component. Use of this tab to provide such information is highly recommended.

Important Incrementing file version numbers is extremely important for components, as it helps ensure that Setup for your component will never overwrite a newer version with an older one.

3

Version Compatibility

On the Component tab, you can select a Version Compatibility mode. For new projects, this option is automatically set to Project Compatibility the first time you compile your component, as discussed in “How to Test ActiveX Components” in “Debugging, Testing, and Deploying Components.”

For successive versions of your component, you can select Binary Compatibility to ensure that programs compiled with old versions continue to work with the new version. See “Version Compatibility” in “Debugging, Testing, and Deploying Components.”

Polymorphism, Interfaces, Type Libraries, and GUIDs

Components deliver services by providing classes from which clients can create objects. Clients use services by creating objects and calling their properties and methods.

Information about the classes provided by your component is contained in a type library. In Visual Basic, the type library is included as a resource in the compiled component. Clients access the type library by setting references to it.

Setting the Type Library Name

Project Name, on the General tab of the Project Properties dialog box, sets the name of your component’s type library, and is used to qualify the names of classes. For example, the following code fragment declares a variable that will hold a reference to an object of the Widget class, provided by a component whose project name is SmallMechanicals:

```
Public gwdgDriveLink As SmallMechanicals.Widget
```

4

Some applications can manipulate objects, but cannot declare variables of a specific object type. Such applications declare generic object variables As Object, and use the

project name in the *projectname* argument of the CreateObject function to get a new object reference, as shown in the following syntax:

Set *objectvariable* **= CreateObject(**"*projectname.class*"**)**

The combination of project name and class name is sometimes referred to as a *fully qualified class name*, or as a *programmatic ID*. The fully qualified class name may be required to correctly identify an object as belonging to your component. For example, you might implement a Window class in your component. Microsoft Excel also provides a Window object, which could lead to the following confusion for client applications:

```
' A variable of the Microsoft Excel Window class.
Dim xlWindow As Excel.Window
' A variable of the ProgramX component's Window class.
Dim pxWindow As ProgramX.Window
' A variable of the Window class that belongs to the
'   component - Microsoft Excel or ProgramX - that
'   appears first in the client application's
'   References dialog box!
Dim xWindow As Window
```

Default Interfaces

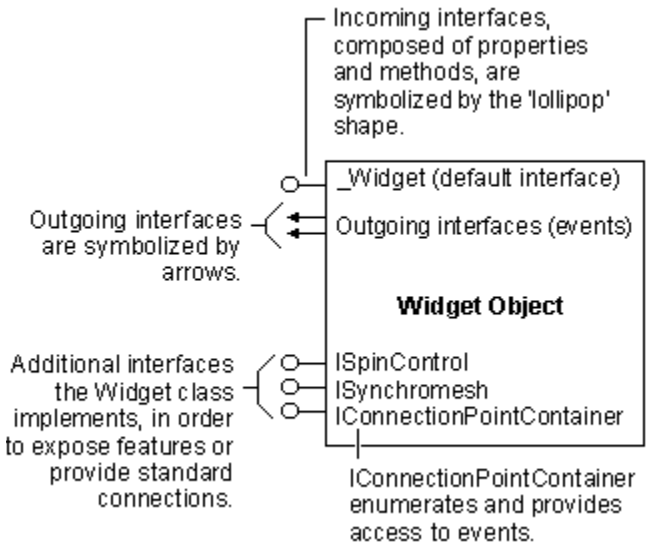
An interface is a set of properties and methods, or events. Every class provided by your component has at least one interface, called the *default interface*, which is composed of all the properties and methods you declare in the class module.

The default interface is usually referred to by the same name as the class, though its actual name is the class name preceded by an underscore. The underscore prefix is a convention, signifying that the name is hidden in the type library.

If the class raises events, it also has an IConnectionPointContainer interface that enumerates those events. Events are outgoing interfaces, as opposed to the incoming interfaces composed of properties and methods. In other words, clients make requests by calling *into* your class's properties and methods, while the events raised by your class call *out* to event handlers in clients.

Incoming and outgoing interfaces are symbolized differently in interface diagrams, as shown in Figure 6.1.

Figure 6.1 Incoming and outgoing interfaces



6

Important You should consider interfaces as contracts between you and the user of your component, because changing an interface may cause applications compiled against your component to fail.

7

Visual Basic provides two mechanisms for enhancing components without affecting compiled applications: version compatibility for interfaces, and multiple interfaces. In order to discuss these mechanisms, however, you have to learn to dig GUIDs.

Type Libraries, Interfaces, and GUIDs

GUID (pronounced *goo*-id) stands for Globally Unique IDentifier, a 128-bit (16-byte) number generated by an algorithm designed to ensure its uniqueness. This algorithm is part of the Open Software Foundation (OSF) Distributed Computing Environment (DCE), a set of standards for distributed computing.

GUIDs are used to uniquely identify entries in the Windows registry. For example, Visual Basic automatically generates a GUID that identifies your type library in the Windows registry.

Visual Basic also automatically generates a GUID for each public class and interface in your component. These are usually referred to as *class IDs* (CLSID) and *interface IDs* (IID). Class IDs and interface IDs are the keys to version compatibility for components authored using Visual Basic.

Note You may also see GUIDs referred to as UUIDs, or Universally Unique IDentifiers.

8

What If Visual Basic Runs Out of GUIDs?

This is not a problem we need to worry about in our lifetimes. The algorithm that generates GUIDs would allow you to compile several new versions of your component every second for centuries — without repeating or colliding with GUIDs generated by other developers.

Version Compatibility for Interfaces

When a developer compiles a program that uses your component, the class IDs and interface IDs of any objects the program creates are included in the executable.

The program uses the class ID to request that your component create an object, and then queries the object for the interface ID. An error occurs if the interface ID no longer exists.

During development of a new component Visual Basic generates new CLSIDs and IIDs every time you compile, as long as either Project Compatibility or No Compatibility is selected on the Component tab of the Project Properties dialog box. Once you've released a component, however, and begin working on an enhanced version of it, you can use the Binary Version Compatibility feature of Visual Basic to change this behavior.

As described in detail in "Version Compatibility" in "Debugging, Testing, and Deploying Components," binary version compatibility preserves the class IDs and interface IDs from previous versions of your component. This allows applications compiled using previous versions to work with the new version.

To ensure compatibility, Visual Basic places certain restrictions on changes you make to default interfaces. Visual Basic allows you to add new classes, and to enhance the default interface of any existing class by adding properties and methods. Removing classes, properties, or methods, or changing the arguments of existing properties or methods, will cause Visual Basic to issue incompatibility warnings.

If you decide to ship an incompatible interface, Visual Basic changes the major version number of the type library and suggests that you change the executable name and Project Name, so that the new version of your component will not over-write the old on your users' hard disks.

Multiple Interfaces: Polymorphism and Compatibility

The Implements statement, discussed in "Polymorphism," in "Programming with Objects", allows your classes to implement additional interfaces.

When two classes implement the same secondary interface, they are said to be *polymorphic* with respect to that interface. That is, a client can make early-bound calls to the properties and methods of the interface without having to know the class of the object it's using.

By creating standard interfaces and implementing them in multiple classes, provided by one or more components, you can take advantage of polymorphism in your applications, or across your entire organization.

For More Information "Creating Standard Interfaces with Visual Basic," later in this chapter, explains how to create interfaces by defining class modules that have no implementation code.

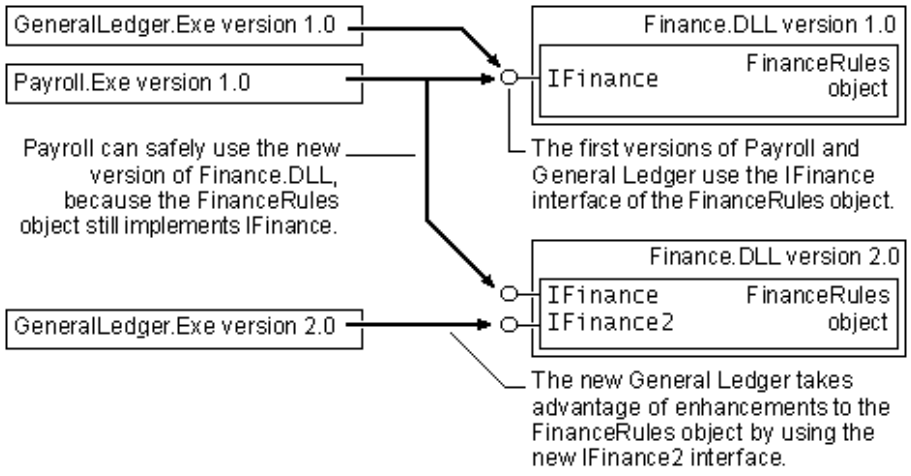
Interfaces and Compatibility

Multiple interfaces provide an alternate means of enhancing and evolving your components while maintaining compatibility with older applications compiled against earlier versions.

The ActiveX rule you must follow to ensure compatibility with multiple interfaces is simple: *once an interface is in use, it can never be changed.* The interface ID of a standard interface is fixed by the type library that defines the interface.

The way to enhance standard interfaces is to create a new standard interface, embodying the enhancements. Future components, or future versions of existing components, can implement the old interface, the new interface, or both, as shown in Figure 6.2.

Figure 6.2 Providing compatibility with multiple interfaces



7

The IFinance and IFinance2 interfaces are defined in separate type libraries, which are referenced by the components that implement the interfaces (in this case, versions 1.0 and 2.0 of Finance.dll), and by the applications that use the interfaces (Payroll and GeneralLedger).

System evolution is possible because future applications can take advantage of new interfaces. Existing applications will continue to work with new versions of

components, as long as the components continue to implement the old interfaces as well as the new.

Applications that Work With Multiple Versions of Components

You can write applications that can use any of several versions of a component. For example, version 2.0 of GeneralLedger.exe could be written to use IFinance2 if that interface was available, and to use IFinance otherwise.

Obviously, GeneralLedger would be able to provide only a limited set of features in the latter case. This ability to provide limited functionality in the absence of the preferred interface is often referred to as *degrading gracefully*.

To work with either interface, GeneralLedger might contain code like the following:

```
Dim fnr As FinanceRules
Dim ifin As IFinance
Dim ifin2 As IFinance2

On Error Resume Next
Set fnr = New FinanceRules
' (Error handling code omitted.)
' Attempt to access the preferred interface.
Set ifin2 = fnr
If Err.Number <> 0 Then
    ' Access the more limited interface.
    Set ifin = fnr
    ' (Code to provide limited functionality,
    '  using the object variable ifin.)
Else
    ' (Code to provide full functionality,
    '  using the object variable ifin2.)
End If
```

9

For More Information "Providing Polymorphism by Implementing Interfaces," later in this chapter, discusses the use and naming of standard interfaces.

Adding Classes to Components

Only one thing distinguishes a component from other applications you author using Visual Basic: A component project has at least one public class from which client applications can create objects.

Like any other Visual Basic application, your component may have numerous class modules that encapsulate its internal functionality. When you allow clients to create instances of a class, objects created from that class can be manipulated by clients, and your application becomes a component.

Creating New Classes

—10

From the Project menu, you can choose Add Class Module, Add User Control, or Add User Document to define a new public class. Other choices on the Project menu allow you to add objects that can be used within your application, but only UserControls, UserDocuments, and class modules can define public classes.

Each public class you add will be the blueprint for one kind of public object in your object model. You can provide a class name, define interfaces for the class, and set the Instancing property (or the Public property, in some cases) to determine how objects will be created from the class.

Name Property

Choose class names carefully. They should be short but descriptive, and formed from whole words with individual words capitalized — for example, *BusinessRule.*

The class name is combined with the name of the component to produce a fully qualified class name, also referred to as a *programmatic ID* or *ProgID*. For example, the fully qualified class name of a BusinessRule class provided by the Finance component, is Finance.BusinessRule.

The topic "What's in a Name?" earlier in this chapter, outlines the rules for naming classes, properties, and methods.

Defining Interfaces

The default interface for a class is composed of the properties and methods you define for it, as discussed in "Adding Properties and Methods to Classes," later in this chapter.

The default interface of a class is an *incoming interface*, as explained in "Polymorphism, Interfaces, Type Libraries, and GUIDs." You can also add *outgoing interfaces*, or events, as described in "Adding Events to Classes."

Visual Basic includes information about the class module's default interface and outgoing interfaces in the type library it creates when your component is compiled.

For More Information You can implement additional incoming interfaces on a class, as described in "Providing Polymorphism by Implementing Interfaces."

Public or Instancing Property

UserControl classes have a Public property that determines whether the class is public or private. UserDocument classes are always public. This is discussed in the in-depth chapters on ActiveX controls, "Building ActiveX Controls."

Class modules have a more complex public life, controlled by the Instancing property. For each class your component will provide to other applications, set the Instancing property of the class module to any value *except* Private, as discussed in the related topic "Instancing for Classes Provided by ActiveX Components."

You don't have to make all your classes public; if there are objects you want to use only within your component, set the Instancing properties of the class modules that define them to Private. (For a UserControl, set the Public property to False.)

For More Information Class modules in Visual Basic are introduced in "Programming with Objects." Topics specific to classes defined in class modules and ActiveX controls are discussed in depth in "Building Code Components" and "Building ActiveX Controls." Object models are discussed in "Organizing Objects: The Object Model," later in this chapter.

Instancing for Classes Provided by ActiveX Components

The value of the Instancing property determines whether your class is private — that is, for use only within your component — or available for other applications to use.

As its name suggests, the Instancing property also determines how other applications create instances of the class. The property values have the following meanings.

- *Private* means that other applications aren't allowed access to type library information about the class, and cannot create instances of it. Private objects are only for use within your component.
- *PublicNotCreatable* means that other applications can use objects of this class only if your component creates the objects first. Other applications cannot use the CreateObject function or the New operator to create objects from the class.
- *MultiUse* allows other applications to create objects from the class. One instance of your component can provide any number of objects created in this fashion.

  8An out-of-process component can supply multiple objects to multiple clients; an in-process component can supply multiple objects to the client and to any other components in its process.
- *SingleUse* allows other applications to create objects from the class, but every object of this class that a client creates starts a new instance of your component. Not allowed in ActiveX DLL projects.

8

The following table shows how values of the Instancing property in Visual Basic 5.0 were formerly expressed by the combination of the Public and Instancing properties in Visual Basic 4.0.

5.0 Instancing	4.0 Public	4.0 Instancing
Private	False	(Any value)
PublicNotCreatable	True	Not Createable
MultiUse	True	Createable MultiUse
SingleUse	True	Createable SingleUse

10

Class Modules and Project Types

The value of the Instancing property is restricted in certain project types. Allowed values are shown in the following table:

Instancing Value	ActiveX EXE	ActiveX DLL	ActiveX Control
Private	Yes	Yes	Yes
PublicNotCreatable	Yes	Yes	Yes
MultiUse	Yes	Yes	
SingleUse	Yes		

11

Dependent Objects (PublicNotCreatable)

The value of the Instancing property determines the part an object plays in your component's object model, as discussed in "Organizing Objects: The Object Model."

If the Instancing property of a class is PublicNotCreatable, objects of that class are called *dependent objects.* Dependent objects are typically parts of more complex objects.

For example, you might allow a client application to create multiple Library objects, but you might want Book objects to exist only as parts of a Library. You can make the Book class PublicNotCreateable, and let the user add new books to a Library object by giving the Library class a Books collection with an Add method that creates new books only within the collection.

Your component can support as many dependent objects as necessary. You can write code in the Add method of a collection class to limit the number of objects in the collection, or you can allow the number to be limited by available memory.

For More Information Dependent objects are discussed in detail in "Dependent Objects," later in this chapter.

12

Externally Creatable Objects

All values of the Instancing property besides PublicNotCreatable and Private define externally creatable objects — that is, objects that clients can create using the New operator or the CreateObject function. ActiveX control projects cannot provide externally creatable objects.

Coding Robust Initialize and Terminate Events

Classes you define in Visual Basic, whether class modules, UserControls, or UserDocuments, have built-in Initialize and Terminate events. The code you place in the Initialize event will be the first code executed when an object is created.

For example, in the first part of the following code fragment the Widget class of the SmallMechanicals component sets the value of its read-only Created property at the moment an object is created.

```
' Code for the component's Widget class module.
```

—13

```
' Storage for the read-only Created property.
Private mdatCreated As Date

' Implementation of the read-only Created property.
Public Property Get Created() As Date
    Created = mdatCreated
End Property

' Set the value for the read-only Created property when
' the object is created.
Private Sub Class_Initialize()
    mdatCreated = Now
End Sub

' Code for the client application.
Private Sub cmdOK_Click()
    Dim wdgX As New SmallMechanicals.Widget
    ' Display date/time object was created.
    MsgBox wdgX.Created
End Sub
```

13

In the last part of the code fragment, the client creates a Widget object. The variable wdgX will contain the reference to the Widget object; it is declared As New, so the Widget is created at the first use of wdgX in code. When the MsgBox function is executed, the Widget is created, and the very first code it executes is its Class_Initialize event procedure. When the read-only Created property of the newly created Widget is evaluated, its value has already been set, and therefore MsgBox correctly displays the time the Widget was created.

Errors that occur in the Class_Initialize event procedure are returned to the point in the client at which the object was requested. Thus, adding the following line to the Widget's Class_Initialize event procedure will cause error 31013 to occur on the client's MsgBox statement.

```
    Err.Raise Number:=31013
```

14

Handling Errors in the Terminate Event

The Terminate event is the last event in an object's life. You can place cleanup code in the Class_Terminate event procedure, and this code will be executed when the last reference to the object has been released, and the object is about to be destroyed. Complex objects that contain dependent objects should release references to their dependent objects in the Terminate event.

Errors in the Terminate event require careful handling. Because the Terminate event is not called by the client application, there is no procedure above it on the call stack. *This means that an unhandled error in a Terminate event will cause a fatal error in the component.*

Important For in-process components, your fatal error is your client's fatal error. Because the component is running in the client's process, the client application will be terminated by a component's fatal error.

—14

15

Standard Modules vs. Class Modules

Classes differ from standard modules in the way their data is stored. There is never more than one copy of a standard module's data. This means that when one part of your program changes a public variable in a standard module, and another part of your program subsequently reads that variable, it will get the same value.

Class module data, on the other hand, exists separately for each instance of the class.

Avoid making the code in your classes dependent on global data — that is, public variables in standard modules. Many instances of a class can exist simultaneously, and all of these objects share the global data in your component.

Static Class Data

Using global variables in class module code violates the object-oriented programming concept of encapsulation, because objects created from such a class do not contain all their data. However, there may be occasions when you want a data member to be shared among all objects created from a class module. For example, you might want all objects created from a class to share a property value, such as the name or version number of your component.

This deliberate violation of encapsulation is sometimes referred to as *static class data*. You can implement static class data in a Visual Basic class module by using Property procedures to set and return the value of a Public data member in a standard module, as in the following code fragment:

```
' Read-only property returning application name.
Property Get ComponentName() As String
    ' The variable gstrComponentName is stored in a
    '   standard module, and declared Public.
    ComponentName = gstrComponentName
End Property
```

16

You can implement static class data that is not read-only by providing a corresponding Property Let procedure — or Property Set for a property that contains an object reference — to assign a new value to the standard module data member.

Important When designing a class that uses static data, remember that your component may be providing objects simultaneously to several client applications (if it's an out-of-process component) or to a client and several in-process components (if it's an in-process component). All the objects created from the class will share the static data, even if they're being used by different clients.

17

Adding Properties and Methods to Classes

The default interface of a class in your component is simply the set of all public properties, methods, and events in the class module, UserControl, or UserDocument that defines the class.

Adding properties and methods is easy — a method is any Public Sub or Public Function procedure you declare in the module that defines your class; a property is any public property procedure or public variable you declare.

For More Information The mechanics of property and method declaration are discussed in "Adding Properties to a Class" and "Adding Methods to a Class," in "Programming with Objects."

18

Implementing Properties in Components

Adding Properties to a Class," in "Programming with Objects," discusses in detail the many kinds of properties you can add to your classes, including simple data values, read-only properties, and property arrays.

"Adding Properties to a Class" also describes the two ways you can declare properties: as public variables, or as property procedures.

In general, properties of objects provided by components should be implemented as property procedures. Property procedures are more robust than data members. A property whose type is an enumeration, for example, cannot be validated unless implemented as a Property Get and Property Let.

The only exception to this rule is a simple numeric or string property which requires no validation and which, when changed, does not immediately affect other properties of the object.

An object property — that is, any property that contains an object reference instead of an ordinary data type — should almost always be implemented with property procedures. An object property implemented as a public object variable can be set to Nothing accidentally, possibly destroying the object. This is discussed in "Organizing Objects: The Object Model," later in this chapter.

Note Internally, Visual Basic generates a pair of property procedures for every public variable you declare. For this reason, declaring public variables doesn't provide any size or performance benefits.

19

For More Information See "Adding Properties to a Class," in "Programming with Objects."

20

Implementing Methods in Components

When you declare a method, declare all of its arguments as explicit data types whenever possible. Arguments that take object references should be declared as specific class types — for example, As Widget instead of As Object or As Variant.

Strongly typed arguments allow many user errors to be caught by the compiler, rather than occurring only under run-time conditions. The compiler always catches errors, while run-time testing is only as good as the test suite coverage.

This is as true of optional parameters as it is of the method's fixed parameters. For example, the Spin method of a hypothetical Widget object might allow either direct specification of spin direction and speed, or specification of another Widget object from which angular momentum is to be absorbed:

```
Public Sub Spin( _
    Optional ByVal SpinDirection As Boolean = True, _
    Optional ByVal Torque As Double = 0, _
    Optional ByVal ReactingWidget As Widget = Nothing)
    ' (Code to ensure that a valid combination of
    '  arguments was supplied.)
    ' (Implementation code.)
End Sub
```

21

For More Information See "Adding Methods to a Class," in "Programming with Objects."

22

Data Types Allowed in Properties and Methods

Classes can have properties and methods of any public data type supported by Automation. This includes all arguments of properties and methods, as well their return values. The allowed data types include:

- Public objects provided by another component, such as DAO or a component authored using Visual Basic.
- Public objects provided by Visual Basic for applications, such as the Error and Collection objects.
- Objects defined in public classes in the component.
- Public enumerations declared in public class modules.
- Standard system data types defined by Automation, such as OLE_COLOR and OLE_TRISTATE.
- The intrinsic data types provided by Visual Basic.

9

On the Evils of Returning Private Objects

The following data types are not allowed, and references to them should never be returned to client applications:

- All of the objects provided in the Visual Basic (VB) object library — for example, controls. Use the Object Browser to view the entire list.
- All forms.
- All class modules whose Instancing property is set to Private.
- References to ActiveX controls.
- User-defined types.

10

It is possible to trick Visual Basic and pass private objects to client programs. Don't do this. *References to private objects will not keep a component running.*

If your component shuts down, because all references to your public objects have been released, any remaining private objects will be destroyed, *even if clients still hold references to them*.

Subsequent calls to the properties and methods of these objects will cause errors, in the case of out-of-process components. In the case of in-process components, a fatal program fault may occur in the client.

Important Private objects are private for a reason, usually because they were not designed to be used outside your project. Passing them to a client may decrease program stability and cause incompatibility with future versions of Visual Basic. If you need to pass a private class of your own to a client, set the Instancing property to a value other than Private.

23

Choosing a Default Property or Method for a Class

You can mark the most commonly used property or method of a class as the default method. This allows the user of a class to invoke the member without naming it.

To set a property or method as the default

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

6 Click **Advanced** to expand the **Procedure Attributes** dialog box.

7 In the **Name** box, select the property or method that is currently the default for the class. If the class does not currently have a default member, skip to step 5.

3**Note** You can use the Object Browser to find out what the current default member of a class is. When you select the class in the Class list, you can scroll through the members in the Members list; the default member will be marked with a small blue circle beside its icon.

24

8 In the **Procedure ID** box, select **None** to remove the default status of the property or method.

9 In the **Name** box, select the property or method you want to be the new default.

10 In the **Procedure ID** box, select **(Default)**, then click **OK**.

Important A class can have only one default member. If a property or method is already marked as the default, you must reset its procedure ID to None before making another property or method the default. No compile errors will occur if two members are marked as default, but there is no way to predict which one Visual Basic will pick as the default.

Adding Events to Classes

You can add events to any class in your component. Events declared in classes provided by your component can be handled by clients regardless of whether your component is running in process or out of process. All events are public.

You declare an event using the Event keyword:

```
Event SomethingHappened( ByVal HowMuch As Double, _
        ByVal When As Date)
```

You raise the event from within your class module's code, whenever the circumstances that define the event occur.

```
If blnSomethingHappened Then
    RaiseEvent SomethingHappened(dblPriceIncrease, _
        Now)
End If
```

When the event is raised in an instance of the class, code in the SomethingHappened event procedures of any clients that are handling the event *for that particular object* will be executed. Events must be handled on an object-by-object basis; a client cannot elect to handle an event for all currently existing objects of a particular class.

If multiple clients have references to the same object, and are handling an event it raises, control will not return to your component until all clients have processed the event.

You can allow clients to respond to events by declaring a parameter ByRef instead of ByVal. This allows any client to change the value of the argument. When execution resumes, on the line after RaiseEvent, you can examine the value of this argument and take appropriate action.

This capability is frequently used for Cancel arguments, as with the QueryUnload event of Visual Basic forms.

Note Visual Basic raises a separate QueryUnload event for each form; if one form cancels the event, events for subsequent forms are not raised.

Events cannot be handled within the class that declared them.

For More Information Raising events in controls is discussed in detail in "Building ActiveX Controls." The syntax for raising and handling events is covered in "Adding Events to Classes" in "Programming with Objects."

29

Providing Named Constants for Your Component

Enumerations provide an easy way to define a set of related named constants. For example, the built-in enumeration VbDayOfWeek contains numeric constants with the names vbMonday, vbTuesday, and so on.

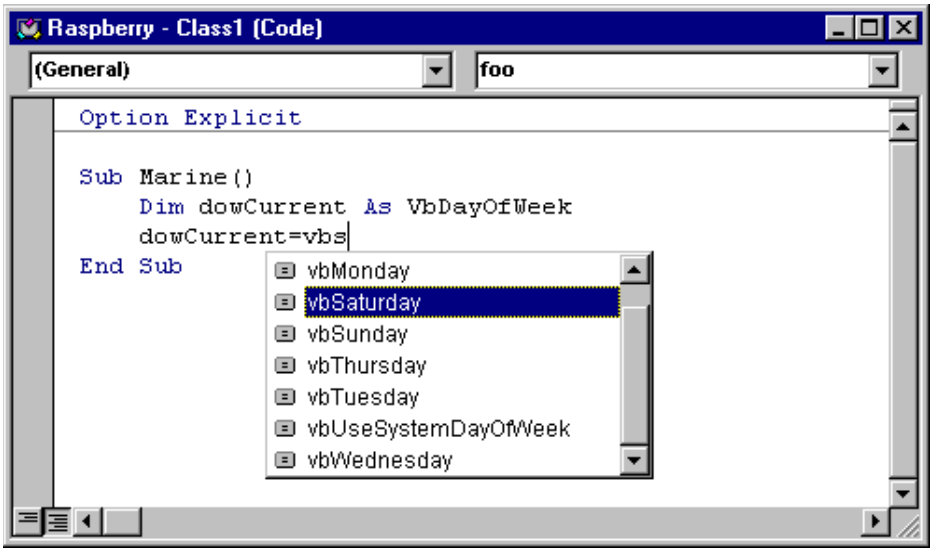
You can use an enumeration as the data type of a property or method argument, as in the following example:

```
Private mdowDayOfWeek As VbDayOfWeek

Property Get DayOfWeek() As VbDayOfWeek
    DayOfWeek = mdowDayOfWeek
End Property

Property Let DayOfWeek(ByVal NewDay As VbDayOfWeek)
    If (NewDay < vbUseSystemDayOfWeek) _
            Or (NewDay < vbSaturday) Then
        Err.Raise Number:=31013, _
            Description:="Invalid day of week"
    Else
        DayOfWeek = mdowDayOfWeek
    End If
End Property
```

30

When users of your component enter code that assigns a value to this property, the Auto List Members feature will offer a drop down containing the members of the enumeration, as shown in Figure 6.3.

Figure 6.3 Auto List Members displays enumerations



12

Tip You might think that you could save space by declaring the internal variable mdowDayOfWeek As Byte instead of As VbDayOfWeek — since the latter effectively makes the variable a Long. However, on 32-bit operating systems the code to load a Long is faster and more compact than the code to load shorter data types. Not only could the extra code exceed the space saved, but there might not be any space saved to begin with — because of alignment requirements for modules and data.

31

You can make the members of an enumeration available to users of your component by marking the enumeration Public and including it in any public module that defines a class — that is, a class module, UserControl, or UserDocument.

When you compile your component, the enumeration will be added to the type library. Object browsers will show both the enumeration and its individual members.

Note Although an enumeration must appear in a module that defines a class, it always has global scope in the type library. It is not limited to, or associated in any other way with the class in which you declared it.

32

General Purpose Enumerations

The members of an enumeration need not be sequential or contiguous. Thus, if you have some general-purpose numeric constants you wish to define for your component, you can put them into a catch-all Enum.

```
Public Enum General
    levsFeetInAMile = 5280
    levsIgnitionTemp = 451
```

```
    levsAnswer = 42
End Enum
```

33

Avoiding Enumeration Name Conflicts

In the preceding code example, both the Enum and its members were prefixed with four lowercase characters chosen to identify the component they belong to, and to reduce the chance that users of the component will encounter name conflicts. This is one of the general naming rules discussed in "What's in a Name?" earlier in this chapter.

For More Information Enumerations are discussed in detail in "More About Programming" and "Programming with Objects."

34

Providing Non-Numeric and Non-Integer Constants

The members of an Enum can have any value that fits in a Long. That is, they can assume any integer value from -2,147,483,648 to 2,147,483,647. When you declare a variable using the name of an Enum as the data type, you're effectively declaring the variable As Long.

Occasionally you may need to provide a string constant, or a constant that isn't an integer value. Visual Basic doesn't provide a mechanism for adding such values to your type library as public constants, but you can get a similar effect using a global object with read-only properties.

If your component doesn't contain a global object, such as Application, add a public class module named GlobalConstants to your project. Set the Instancing property to GlobalMultiUse.

For each constant you want to provide, add to the GlobalConstants class module a Property Get procedure that returns the desired value. For example, the following code provides Avogadro's Number as a constant, and mimics the vbCrLf constant in Visual Basic.

```
Public Property Get Avogadro() As Double
    Avogadro = 6.02E+23
End Property

Public Property Get vbCrLf() As String
    vbCrLf = Chr$(13) & Chr$(10)
End Property
```

35

Because the Instancing property is GlobalMultiUse, a user of the component doesn't have to explicitly create an instance of the GlobalConstants class in order to use the constants. The constants can be used as if they were part of Visual Basic:

```
    strNewText = "Line1" & vbCrLf & "Line2"
```

36

Note A user of Visual Basic, Microsoft Excel, or any other application that hosts Visual Basic for Applications would never see this version of the vbCrLf constant, because the VBA type library is always higher in the References dialog than the type library of any component.

37

Private Communications Between Your Objects

There may be circumstances in which you want your component's objects to be able to communicate with each other, without interference from users of your component. For example, you might want your Widgets collection class to set the Parent property of a newly created Widget, and thereafter to have Parent be read-only.

Public methods on a class can be called by other objects, but they can also be called by clients. Private methods cannot be called from outside the component, but neither are they visible to other objects within your component.

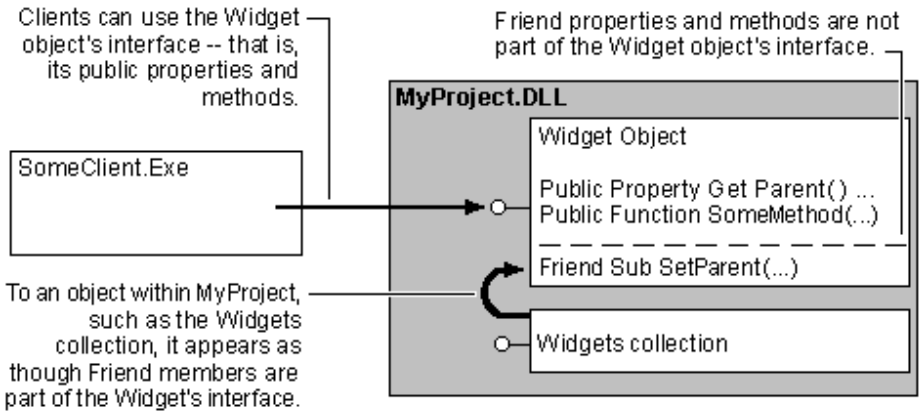
The solution is to use Friend methods. In the following code fragment, the hypothetical Widget object exposes a public read-only Parent property, and a Friend method (called SetParent) that the Widgets collection can use to set the value of the Parent property after creating a new Widget.

```
' A Widget is always part of a mechanism.
Private mmchParent As Mechanism

Public Property Get Parent() As Mechanism
    Set Parent = mmchParent
End Property

Friend Sub SetParent(ByVal NewParent As Mechanism)
    Set mmchParent = NewParent
End Sub
```

38

When a method is declared with the Friend keyword, it's visible to other objects in your component, but is not added to the type library or the public interface. This is illustrated in Figure 6.4.

Figure 6.4 Friend methods have project scope



13

At run time, the Widgets collection class (within the project) sees a different interface from that seen by clients. The view of the Widget's interface within the project (and the compiled DLL) includes the Friend method SetParent, which the Widgets collection calls.

The client only sees the public properties and methods of the Widget's interface, because Friend methods are not added to the type library.

Using the Friend Keyword with Properties

You can also declare property procedures with the Friend keyword. In fact, the different property procedures that make up a property can have different scope. Thus the earlier code example can be rewritten as a pair of property procedures:

```
' A Widget is always part of a mechanism.
Private mmchParent As Mechanism

Public Property Get Parent() As Mechanism
    Set Parent = mmchParent
End Property

Friend Property Set Parent(ByVal NewParent As _
        Mechanism)
    Set mmchParent = NewParent
End Sub
```

39

From within the component, Parent is a read/write property. To clients, it's a read-only property, because only the Property Get appears in the component's type library.

You can think of Friend as a different scope, halfway between Public and Private.

Important In order to invoke Friend methods and properties, you must use strongly typed object variables. In the example above, the Widgets collection

must use a variable declared As Widget in order to access the SetParent method or the Property Set Parent. You cannot invoke Friend methods from variables declared As Object.

40

Hiding Object Properties that Return Private Objects

"Using Properties and Collections to Create Object Models" describes the use of private objects in object models. When linking such objects to the public objects in the object model, you can declare all parts of the property procedure using the Friend keyword.

For example, each Widget object might have Socket object, which for some reason you don't want to expose to users of your component. You could add the following object property to the Widget object, so that from inside your component you could access the Socket, without adding the property to the type library or the public interface:

```
' Create the Socket object on demand (As New).
Private msoc As New Socket

Friend Property Get Socket() As Socket
    Set Socket = msoc
End Property
```

41

For More Information Friend methods are introduced in "Programming with Objects."

42

Providing Polymorphism by Implementing Interfaces

One of the most striking features of the Component Object Model (COM) is the ability of an object to implement multiple interfaces. In addition to enabling polymorphism, multiple interfaces provide a mechanism for incremental or evolutionary development, without the need to recompile all the components in the system when changes occur.

By defining features in terms of interfaces, composed of small groups of closely-related functions, you can implement component features as needed, knowing that you can expand them later by implementing additional interfaces.

Maintaining compatibility is simplified, because new versions of a component can continue to provide existing interfaces, while adding new or enhanced interfaces. Succeeding versions of client applications can take advantage of these when it makes sense for them to do so.

Inheritance and Polymorphism

As explained in “Polymorphism” in “Programming with Objects”, most object-oriented programming tools provide polymorphism through inheritance. This is a powerful mechanism for small-scale development tasks, but has generally proven to be problematic for large-scale systems.

In part, these difficulties arise as a result of necessary changes to classes deep in the inheritance tree. Recompilation is required in order to take advantage of such changes, and failure to recompile may lead to unpleasant surprises when the time finally arrives for a new version.

More seriously, an over-emphasis on inheritance-driven polymorphism typically results in a massive shift of resources from development tasks to up-front design tasks, doing nothing to address development backlogs or to shorten the time before the end user can discover — through hands-on experience — whether the system actually fulfills the intended purpose.

As a consequence, tools for rapid prototyping and Rapid Application Development (RAD) have gained wider acceptance than OOP tools.

Visual Basic and COM

Visual Basic follows the COM example, emphasizing multiple interfaces as a more flexible way to provide polymorphism. Software can evolve interface by interface, rather than having to be derived from all necessary antecedents during a lengthy design process.

Objects can begin small, with minimal functionality, and over time acquire additional features, as it becomes clear from actual use what those features should be. Legacy code is protected by continuing to support old interfaces while implementing new ones.

The Implements Feature

Visual Basic provides the Implements keyword as the means for incorporating a secondary interface. For example, if your project had a reference to a type library that described the IFinance interface, you could place the following code in a class module:

```
Implements IFinance
```

43

Because type libraries contain only interfaces, and no implementation, you would then add code for each of the properties and methods of the IFinance interface, as described in “Implementing and Using Standard Interfaces.”

An Interface is a Contract

When you create an interface for use with Implements, you’re casting it in concrete for all time. This *interface invariance* is an important principle of component design, because it protects existing systems that have been written to an interface.

When an interface is clearly in need of enhancement, a new interface should be created. This interface might be called Interface2, to show its relationship to the existing interface.

While generating new interfaces too frequently can bulk up your components with unused interfaces, well-designed interfaces tend to be small and independent of each other, reducing the potential for performance problems.

Factoring Interfaces

The process of determining what properties and methods belong on an interface is called *factoring*.

In general, you should group a few closely-related functions on an interface. Too many functions make the interface unwieldy, while dividing the parts of a feature too finely results in extra overhead. For example, the following code calls methods on three different interfaces of the Velociraptor class:

```
Public Sub CretaceousToDoList(ByVal vcr1 As _
        Velociraptor, ByVal vcr2 As Velociraptor)
    Dim dnr As IDinosaur
    Dim prd As IPredator
    vcr1.Mate vcr2
    Set dnr = vcr1
    dnr.LayEggs
    Set prd = vcr1
    prd.KillSomethingAndEatIt
End Sub
```

44

In order to use methods on the IDinosaur and IPredator interfaces, you must assign the object to a variable of the correct interface type.

Where possible, interfaces designed to use flexible data structures will outlast interfaces based on fixed data types.

As noted above, it's much harder to go wrong in designing interfaces than in creating large inheritance trees. If you start small, you can have parts of a system running relatively quickly. The ability to evolve the system by adding interfaces allows you to gain the advantages object-oriented programming was intended to provide.

For More Information The Implements feature is discussed in detail in "Polymorphism" in "Programming with Objects."

45

Creating Standard Interfaces with Visual Basic

You can create standard interfaces for your organization by compiling abstract classes in Visual Basic ActiveX DLLs or EXEs, or with the MkTypLib utility, included in the Tools directory.

The MkTypLib utility may be more comfortable for you if you're an experienced user of Microsoft Visual C++.

Basic programmers may find it easier to create an interface using a Visual Basic class module. Open a new ActiveX DLL or EXE project, and add the desired properties and methods to a class module. Don't put any code in the procedures. Give the class the name you want the interface to have, for example IFinance, and make the project.

Note The capital "I" in front of interface names is an ActiveX convention. It is not strictly necessary to follow this convention. However, it provides an easy way to distinguish between abstract interfaces you've implemented and the default interfaces of classes. The latter are usually referred to by the class name in Visual Basic.

46

The type library in the resulting .dll or .exe file will contain the information required by the Implements statement. To use it in another project, use the Browse button on the References dialog box to locate the .dll or .exe file and set a reference. You can use the Object Browser to see what interfaces a type library contains.

Important The Implements feature does not support outgoing interfaces. Thus, any events you declare in the class module will be ignored.

47

As explained in "Providing Polymorphism by Implementing Interfaces," an interface once defined and accepted must remain invariant, to protect applications written to use it. *DO NOT* use the Version Compatibility feature of Visual Basic to alter standard interfaces.

For More Information The related topic "Providing Polymorphism by Implementing Interfaces" discusses such important concepts as interface invariance and factoring. "Implementing and Using Standard Interfaces" explains how interfaces are implemented and used in components.

48

Implementing and Using Standard Interfaces

Once you've defined a standard interface, either by creating a type library with the MkTypLib utilityor by compiling a Visual Basic project containing abstract classes (that is, class modules with properties and methods that don't contain any code), you can implement that interface in classes your components provide.

Suppose you had a LateCretaceous system that included a number of components, each of which provided objects representing flora, fauna, and business rules of that era. For example, one component might provide a Velociraptor class, while another provided a Tyrannosaur class.

You might create a standard interface named IPredator, which included Hunt and Attack methods:

```
' Code for the abstract IPredator class module.
Public Sub Hunt()

End Sub
```

```
Public Sub Attack(ByVal Victim As IDinosaur)

End Sub
```

49

Notice that the argument of the Attack method uses another interface, IDinosaur. One would expect this interface to contain methods describing general dinosaur behavior, such as laying eggs, and that it would be implemented by many classes — Velociraptor, Tyrannosaur, Brontosaur, Triceratops, and so on.

Notice also that there's no code in these methods. IPredator is an *abstract class* that simply defines the interface (referred to as an *abstract interface*). Implementation details will vary according to the object that implements the interface.

For example, the Tyrannosaur class might implement IPredator as follows:

```
Implements IPredator

Private Sub IPredator_Hunt()
    ' Code to stalk around the landscape roaring, until
    '   you encounter a dinosaur large enough to
    '   qualify as a meal.
End Sub

Private Sub IPredator_Attack(ByVal Victim As IDinosaur)
    ' Code to charge, roaring and taking huge bites.
End Sub
```

50

Important As noted in "Providing Polymorphism by Implementing Interfaces," an interface is a contract. You must implement *all* of the properties and methods in the interface.

51

By contrast, the Velociraptor class might implement IPredator as shown here:

```
Implements IPredator

Private Sub IPredator_Hunt()
    ' Fan out and hunt with a pack, running down
    '   small dinosaurs or surrounding large ones.
End Sub

Private Sub IPredator_Attack(ByVal Victim As IDinosaur)
    ' Code to dart in from all sides, slashing the
    '   victim and wearing it down.
End Sub
```

52

Using Implemented Interfaces

Once you have classes that implement IPredator, you can upgrade your existing applications one by one to use the new, more competitive interface. You can access the Hunt and Attack methods by assigning a Velociraptor or Tyrannosaur object to a variable of type IPredator, as shown here:

```
Dim tyr As New Tyrannosaur
Dim prd As IPredator
```

```
    Set prd = tyr
    prd.Hunt
```

53

You can also declare procedure arguments As IPredator, and pass the procedure any object that implements the IPredator interface, as here:

```
Public Sub DevourTheCompetition(ByVal Agent As _
        IPredator, ByVal Target As IDinosaur)
    Agent.Hunt
    Agent.Attack Target
End Sub
```

54

The Sub procedure shown above could be called with any predatory dinosaur as the first argument, and any dinosaur at all as the second. The caller of the procedure can use whatever predatory dinosaur is most appropriate for the occasion. This kind of flexibility is important in maintaining a business advantage.

Setting References to Type Libraries

A type library containing abstract interfaces provides a reference point for both implementing and using interfaces.

In order to implement an interface, you must use the References dialog box to set a reference to the type library. This is because the type library contains the information required to specify the arguments and return types of the interface's members.

In similar fashion, any application that uses objects which have implemented an abstract interface must also have a reference to the type library that describes the interface. Information about implemented interfaces cannot be included in the type libraries of components, because there is no way to resolve naming conflicts.

Important In order to marshal data between processes or between remote computers, out-of-process components must include in their Setup programs any type libraries that describe abstract interfaces. In-process components should also include these type libraries, because a developer may want to pass its objects to other applications, either on the local computer or on a remote computer. See "Deploying Components," in "Debugging, Testing, and Deploying Components."

55

Summary

The following list provides an outline for implementing multiple interfaces:

13. Define a set of interfaces, each containing a small group of related properties and methods that describe a service or feature your system requires. This *factoring* process is discussed in "Providing Polymorphism by Implementing Interfaces."
14. Create a type library containing abstract interfaces — abstract classes, if you create the type library by compiling a Visual Basic project — that specify the arguments and return types of the properties and methods. Use the MkTypLib utilityor Visual Basic to generate the type library, as described in "Creating Standard Interfaces with Visual Basic."

15. Develop a component that uses the interfaces, by adding a reference to the type library and then using the Implements statement to give classes secondary interfaces as appropriate.
16. For every interface you've added to a class, select each property or method in turn, and add code to implement the functionality in a manner appropriate for that class. See "Polymorphism" in "Programming with Objects."
17. Compile the component and create a Setup program, making sure you include the type library that describes the abstract interfaces.
18. Develop an application that uses the component by adding references to the component and to the type library that describes the abstract interfaces.
19. Compile the application and create a Setup program, including the component (and the abstract type library, if the component runs out of process or — with the Enterprise Edition — on a remote computer).

14

The next section discusses how the process outlined here can be used to gradually enhance a system.

Systems that Evolve Over Time

The observant reader will no doubt have noticed a bug in the code given earlier in this topic. If predatory dinosaurs only ate other dinosaurs, how did they keep the Mammals down? A more general IPredator interface might accept as a victim any object that implemented IAnimal.

This illustrates a key advantage of component software development using multiple interfaces: As the LateCretaceous system evolves into, say, the Pliestocene system, components that provide predatory dinosaur objects can be replaced by components that provide SaberTooth and DireWolf objects.

A legacy application compiled to use dinosaurs may be still be able to function quite nicely using the new predator classes, as long as it doesn't include code specific to dinosaurs.

The key points to remember when using multiple interfaces in this fashion are:

- Once an interface is defined and in use, it must never change. This concept of *interface invariance* is discussed in "Providing Polymorphism by Implementing Interfaces," in this chapter, and in "Polymorphism" in "Programming with Objects."
- If an interface needs to be expanded, create a new interface. This is discussed in "Polymorphism, Interfaces, Type Libraries, and GUIDs," earlier in this chapter.
- New versions of components can provide new features by implementing new and expanded interfaces.
- New versions of components can support legacy code by continuing to provide old interfaces.

—31

- New versions of applications can take advantage of new features (that is, new and expanded interfaces), and if necessary can be written so as to degrade gracefully when only older interfaces are available. (See “Polymorphism, Interfaces, Type Libraries, and GUIDs.”)

Implements and Code Reuse

The Implements statement also allows you to reuse code in existing objects. In this form of code reuse, the new object (referred to as an *outer object*) creates an instance of the existing object (or *inner object*) during its Initialize event.

In addition to any abstract interfaces it implements, the outer object implements the default interface of the inner object. (To do this, use the References dialog box to add a reference to the component that provides the inner object.)

When adding code to the outer object’s implementations of the properties and methods of the inner object, you can delegate to the inner object whenever the functionality it provides meets the needs of the outer object.

For example, the Tyrannosaur class might implement the interface of a Dinosaur object (instead of an abstract IDinosaur interface). The Dinosaur object might have a LayEggs method, which the Tyrannosaur class could implement by simple delegation:

```
Private dnoInner As Dinosaur

Private Sub Class_Initialize()
    Set dnoInner = New Dinosaur
End Sub

Private Sub Dinosaur_LayEggs()
    ' Delegate to the inner object.
    dnoInner.LayEggs
End Sub
```

56

This is an extremely powerful and flexible way to reuse code, because the outer object can choose to execute its own code before, after, or instead of delegating to the inner object.

For More Information Code reuse with the Implements statement is discussed in more detail in “Polymorphism” in “Programming with Objects.”

Organizing Objects: The Object Model

An *object model* defines a hierarchy of objects that gives structure to an object-based program. By defining the relationships between objects that are part of the program, an object model organizes the objects in a way that makes programming easier.

The public object model of a component is especially important because it’s used by all the programmers who employ the component as part of their applications.

Note Users of C++ or other object-oriented programming languages are used to seeing *class hierarchies.* A class hierarchy describes *inheritance.* That is, it shows how objects are derived from simpler objects, inheriting their behavior. By contrast, object models are hierarchies that describe *containment.* That is, they show how complex objects like Worksheets contain collections of other objects, such as Button, Picture, and PivotTable objects. Object models can be created with Visual Basic, Visual C++, and other tools that support COM and ActiveX.

57

- "Programming with Objects" includes an introduction to object models and a discussion of design considerations for collection classes.

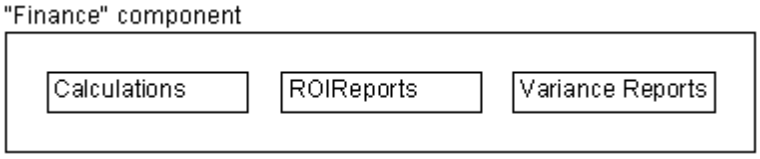
Do I Need an Object Model?

You don't have to create an elaborate object model for your component. A control component (.ocx file) might contain three UserControl objects, and no class modules at all. A code component meant to be used as a simple library of functions might have one global object with a zillion methods.

Again, you might create a code component named Finance with three classes in it, each class representing a self-contained business rule. If the rules are independent of each other, there's no reason to link them into a hierarchy. A client application that uses these rules simply creates one or more objects of each class, as needed.

Each class module in such a component would have its Instancing property set to MultiUse, so that client applications could create objects from the class, and so that the component could handle multiple objects from each class. Figure 6.5 shows such an object model.

Figure 6.5 A flat object model with several externally creatable objects



15

Object Models and Interfaces

As the functionality of an object increases, so does the complexity of its interface. This can make the object hard to use.

You can reduce the complexity of an object's default interface by factoring out groups of related functions, and defining an interface for each group. A client can work with only those interfaces that provide needed features.

By defining standard interfaces in this fashion, you can implement interfaces on other objects, gaining the benefits of polymorphism. This approach to software design in

discussed in “Providing Polymorphism by Implementing Interfaces,” earlier in this chapter.

Sometimes an object is too large even when its features are factored into separate interfaces. When an object becomes very complex, as for example the TreeView and Toolbar controls, breaking pieces of it off as separate objects may make sense.

Once the whole is divided, you need a way of organizing its constituent parts. Splitting the Node object off from the TreeView control gains you nothing if you can’t show the relationship between them. Object models make it easy to provide this organization to the user of your component.

For More Information Topics relating to object models are listed in “Organizing Objects: The Object Model.” The use of multiple interfaces is covered in “Providing Polymorphism by Implementing Interfaces.”

58

Externally Creatable Objects

Part of the additional importance of object models in components comes from the fact that components can provide objects in two different ways—as externally creatable objects or as dependent objects.

In an ordinary program that uses private objects, you can create objects from any class the program defines. A client application, however, can only create objects from some of the classes a component provides. *Externally creatable objects* are those that a client application can create using the New operator with the Set statement, by declaring a variable As New, or by calling the CreateObject function.

When a client uses one of these mechanisms to request an externally creatable object, the component returns a reference the client can use to manipulate the object. When the client sets the last variable containing this reference to Nothing, or allows it to go out of scope, the component destroys the object.

You can make a public object externally creatable by setting the Instancing property of the class module to any value *except* Private or PublicNotCreatable.

For More Information A discussion of the Instancing property can be found in “Instancing for Classes Provided by ActiveX Components.” Dependent objects are discussed in “Dependent Objects.”

59

Dependent Objects

Sometimes there is a clear relationship between two objects, such that one object is a part of the other. In Microsoft Excel, for example, a Button object is always part of another object, such as a Worksheet.

An object that’s contained in another object is called a *dependent object.* Client applications can manipulate dependent objects, just as they can manipulate externally

creatable objects, but they cannot create dependent objects using CreateObject or New.

Set the Instancing property of a class module to PublicNotCreatable to make the objects created from that class dependent objects.

Note Dependent objects are also referred to as *nested objects.*

60

Getting References to Dependent Objects

If a client application can use dependent objects but can't create them, how are they created?

A component can provide dependent objects in several ways. Most commonly an externally creatable object will have a collection with an Add method which the client can invoke. The component creates the dependent object in the code for the Add method, and returns a reference to the new object, which the client can then use.

For example, a Microsoft Excel Worksheet object has a collection of Button objects. A client application can add a new button to the worksheet by calling the Add method of the Buttons collection, as shown in the following code fragment:

```
' Note: The variable wsBudget contains a reference to
'   a Worksheet object.
Dim btnOK As Excel.Button
' Parameters of the Add method specify the top, left,
' width, and height of the new button. The return value
' is a reference to the new Button object.
Set btnOK = wsBudget.Buttons.Add(100, 100, 150, 125)
' Set the caption of the new Button object.
btnOK.Caption = "OK"
```

61

It's important to remember that the variable btnOK contains a reference to the object, not the object itself.

Note The distinction between externally creatable objects and dependent objects is made for the benefit of the client applications that manipulate a component's objects. From *within* a component, you can always create objects from any of the component's classes, regardless of the value of the Instancing property.

62

For More Information "Combining Externally Creatable and Dependent Objects" discusses the process of identifying the types of objects needed for each part of an object model.

63

Combining Externally Creatable and Dependent Objects

The relationships between the externally creatable objects a component provides and the dependent objects they contain are expressed in the component's object model. Once you've analyzed the functionality your component will provide, you can:

20.Determine what objects you need to implement that functionality.

21.List the properties and methods each object will require.

22.Determine the relationships between the objects.

23.Identify the top-level objects that need to be created by client applications.

16

Visual Basic gives you the flexibility to implement many possible object models. A component can provide several unrelated creatable objects, each containing one or more dependent objects; it can also provide a single hierarchy containing a number of objects, only one or two of which are externally creatable.

One characteristic common to all of these implementations is that they require more design time. It's important to spend adequate time and effort determining how your objects will interact and how they will be used designing your object model to avoid having to redefine objects, or split one object into two, in a future version of your component.

Such changes make it much more difficult for applications that use your component to migrate to newer versions. Adding new objects, or exposing objects that were formerly private, does not cause such problems.

For More Information "Using Properties and Collections to Create Object Models" discusses techniques for linking dependent and externally creatable objects in an object model.

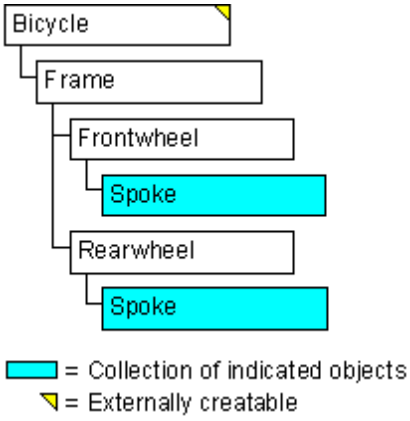
64

Using Properties and Collections to Create Object Models

Objects in a hierarchy are linked together by *object properties*, that is, properties that return references to objects. An object that contains other objects will have properties that return either references to the objects themselves, or references to collections of objects.

For example, consider a Bicycle object that contains two Wheel objects; each Wheel object might in turn contain a Rim object and a collection of Spoke objects.Figure 6.6 shows a possible object model for the externally creatable Bicycle object and its dependent objects.

Figure 6.6 An externally creatable object with a hierarchy of dependent objects



17

The Bicycle object would have a Frame property that returned a reference to its Frame object. The Frame object would have a FrontWheel and BackWheel property, each of which would return a Wheel object. The Wheel object would have a Spokes property that would return a Spokes collection object. The Spokes collection would contain the Spoke objects.

You may also have dependent objects that are used internally by classes in your component, and which you do not want to provide to users of your component. You can set the Instancing properties of the class modules that define these objects to Private, so they won't appear when users browse your type library.

For example, both the Frame and Wheel objects might have collections of Bearing objects, but there may be no reason to expose the Bearings object, or the collections containing it, for manipulation by client applications.

Important In order to keep the Bearings property from appearing in the type library, you must declare it using the Friend keyword, as described in "Private Communications Between Your Objects," earlier in this chapter.

65

The Simple Way to Link Dependent Objects

Frequently it makes sense for a complex object to have only one instance of a dependent object. For example, a Bicycle object only needs one Frame object. In this case, you can implement the linkage as a simple property of the complex object:

```
Private mFrame As Frame

Public Property Get Frame() As Frame
    Set Frame = mFrame
End Property

Private Sub Class_Initialize()
    ' Create the Frame when the Bicycle is initialized.
```

—37

```
    Set mFrame = New Frame
End Sub
```

66

It's important to implement such properties as shown above, using a read-only property procedure, rather than simply declaring a public module-level variable, as shown below.

```
Public Frame As Frame       'Bad idea.
```

67

With the second implementation, a user of your component might set the Frame property to Nothing. If there are no other references to the Frame object, it will be destroyed. The effect of this on the Bicycle object is left to the reader's imagination.

Linking a Fixed Number of Objects

Even when a complex object contains more than one instance of a dependent object, it may make more sense to implement the linkage with properties instead of with a collection. For example, a Bicycle object always has two wheels:

```
' Create the Wheel objects on demand (As New), instead
'   of in the Bicycle object's Initialize event.
Private mwhlFront As New Wheel
Private mwhlRear As New Wheel

Public Property Get FrontWheel() As Wheel
    Set FrontWheel = mwhlFront
End Property

Public Property Get RearWheel() As Wheel
    Set RearWheel = mwhlRear
End Property
```

68

Using Collections in Your Object Model

When the relationship between two objects in a hierarchy is such that the first object contains an indeterminate number of the second, the easiest way to implement the link is with a collection. A *collection* is an object that contains a set of related objects.

For example, the linkage between the FrontWheel object and its Spoke objects in Figure 6.6 is a *collection class*. A collection class is a class module that exists solely to group all the objects of another class. In this case, the collection class is named Spokes, the plural of the name of the class of objects it contains. (See "What's In a Name?", earlier in this chapter, for more information on naming classes.)

Implementing this part of the object model example requires three class modules. From the bottom up, these are:

- The Spoke class module, which defines the properties and methods of a single spoke.
- The Spokes class module, which defines a collection object to contain Spoke objects.

- The Wheel class module, which defines an entire wheel, with a collection of spokes.

18

Dependent Class: Spoke

The Spoke class module is the simplest of the three. It could consist of as little as two Public variables, as in the following code fragment:

```
' Properties for Spoke
Public PartNumber As Integer
Public Alloy As Integer
```

69

The Instancing property of the Spoke class is set to PublicNotCreatable. The *only* way for a client application to create a Spoke object is with the Add method of the Spokes collection, as discussed in the next section.

Note This is not a very robust implementation. In practice you would probably implement both of these properties as Property procedures, with code to validate the values that are assigned to them.

70

For More Information For details on using Property procedures, see "Programming with Objects."

71

Dependent Collection Class: LineItems

The Spokes class module is the template for a collection Spoke objects. It contains a Private variable declared as a Collection object:

```
Private mcolSpokes As Collection
```

72

The collection object is created in the Initialize method for the class:

```
Private Sub Class_Initialize()
    Set mcolSpokes = New Collection
End Sub
```

73

The methods of the Spokes class module *delegate* to the default methods of the Visual Basic Collection object. That is, the actual work is done by the methods of the Collection object. The Spokes class might include the following properties and methods:

```
' Read-only Count property.
Public Property Get Count() As Integer
    Count = mcolSpokes.Count
End Property

' Add method for creating new Spoke objects.
Public Function Add(ByVal PartNumber As Integer, _
                    ByVal Alloy As Integer)
    Dim spkNew As New Spoke
    spkNew.PartNumber = PartNumber
    spkNew.Alloy = Alloy
    mcolSpokes.Add spkNew
    Set Add = spkNew
```

```
End Function
```

74

As with the Spoke class, the Instancing property of the Spokes class is set to PublicNotCreatable. The only way to get a Spokes collection object is as part of a Wheel object, as shown in the following section describing the Wheel class.

For More Information See “Object Models” in “Programming with Objects” for a discussion of collections, including a more detailed explanation of delegation, a list of methods you need to implement, and instructions for creating a collection that works with For Each.

75

Externally Creatable Class: Wheel

The Wheel class module has Instancing set to MultiUse, so that any client application can create a Wheel object. The Wheel class module contains a Private variable of the Spokes class:

```
' Create the Spokes collection object on demand.
Private mSpokes As New Spokes

Public Property Get Spokes() As Spokes
    Set Spokes = mSpokes
End Property
```

76

Every Wheel object a client creates will have its own Spokes collection. The collection is protected against accidentally being set to Nothing by making it a read-only property (Property Get). A developer can access the methods and properties of the Spokes collection as shown in the following code fragment:

```
Dim whl As Wheel
Dim spk As Spoke
Set whl = New Wheel
Set spk = whl.Spokes.Add PartNumber:=3222223, Alloy:=7
' Call a method of the Spoke object.
spk.Adjust
MsgBox whl.Spokes.Count        ' Displays 1 (one item).
```

77

The Add method is used to create a new spoke in the Spokes collection of the Wheel object. The Add method returns a reference to the new Spoke object, so that its properties and methods can be called. A spoke can only be created as a member of the Spokes collection.

The difference between the Wheel object, which can be created by any client, and its dependent objects is the value of the Instancing properties of the classes.

For More Information The Spokes object is created on demand, while the Collection object mcolSpokes was explicitly created. “Programming with Objects” discusses the use of As New for creating variables on demand, including performance implications.

78

Considerations for Linking Objects in an Object Model

Generally speaking, a simpler implementation will be faster. Accessing an item in a collection involves a series of nested references and function calls. Whenever you know that there will always be a fixed number of a dependent object type, you can implement the linkage as a property.

Regardless of how the object model is linked, the key difference between externally creatable objects and dependent objects is the value of the Instancing property of the class module. An object that can be created by other applications will have its Instancing property set to any value except Private or PublicNotCreatable.

All dependent objects, whether they are contained in other dependent objects or in objects that can be created by other applications, will have their Instancing properties set to PublicNotCreatable.

Using Externally Creatable Objects as Dependent Objects

At times you may want to use objects in both ways. That is, you may want the user to be able to create a Widget object independent of the object model, while at the same time providing a Widgets collection as a property of the Mechanism object.

In fact, you may even want to allow the user to create independent instances of the Widgets collection, to move independent Widgets into and out of any Widgets collection, and to copy or move Widgets between collections.

You can make the objects externally creatable by setting the Instancing property of the Widget class and the Widgets collection class to MultiUse.

Important If the Widget object can be created directly by client applications, you cannot depend on all Widget objects getting initialized by the code in the Add method of the Widgets collection. Objects that will be both creatable and dependent should be designed to require no initialization beyond their Initialize events.

79

Allowing free movement of Widgets requires implementation of Insert, Copy, and Move methods for your collection. Insert and Move are fairly straightforward, because moving or inserting a reference to an object is as good as moving the object. Implementing Copy, however, requires more work.

This is because client application never actually has the object in its possession. All the client application has is a reference to an object the component has created on its behalf. Thus, when you implement Copy, you must create a duplicate object, including duplicates of any dependent objects it contains.

For More Information See “Dealing with Circular References” for a discussion of problems that may arise when linking objects together.

80

Dealing with Circular References

Containment relationships allow you to navigate through a hierarchy from a high-level object to any of the objects it contains.

Object models that strictly express containment are like trees. Any given branch (object) may divide into smaller branches (dependent objects), but the smaller branches do not loop around and rejoin the trunk or lower branches.

Object models with loops, or *circular references,* result when a dependent object has a property or variable that holds a reference to one of the objects that contains it.

For example, an Order object might have a Contact property that contains a reference to a Contact object, representing the individual who placed the order. The Contact object might in turn have a Company property that contains a reference to a Company object.

Up to this point, the hierarchy is a tree. However, if the Company object has a MostRecentOrder property that contains a reference to the Order object, a circular reference has been created.

Note You could avoid the circular reference in this case by making the MostRecentOrder property a text key that could be used to retrieve the Order object from the component's Orders collection.

81

Circular References in Visual Basic Components

Consider the simplest form of circular reference, a Parent property. The Parent property of a dependent object contains a reference to the object that contains it.

For example, in the Microsoft Excel object model, a Button object is contained by a Worksheet object. If you have a reference to a Button object, you can print the name of the Worksheet that contains it using code like the following:

```
' If the variable btnCurrent contains a reference to a
' Microsoft Excel Button object, the following line of
' code displays the Name property of the Worksheet
' object that contains the button.
MsgBox btnCurrent.Parent.Name
```

82

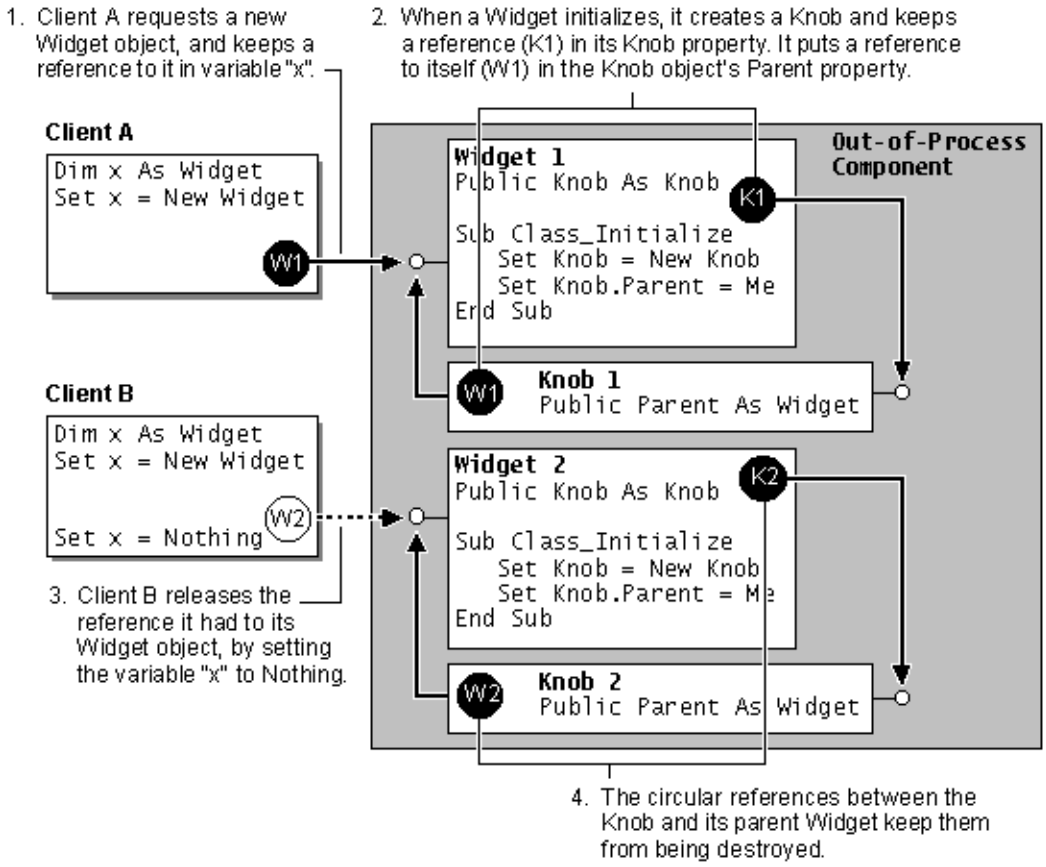
Microsoft Excel is written using C++ and low level COM interfaces, and it maintains the Parent properties of its objects without creating circular references. If you implement such a relationship in Visual Basic, you have to take into account the way Visual Basic handles the creation and destruction of objects.

Visual Basic destroys an object when there are no longer any references to it. If an object's parent has a collection that contains a reference to the object, that's enough to keep the object from being destroyed. In the same way, an object continues to exist if the parent has an object property that contains a reference to the object.

When a parent object is destroyed, the variables that implement its properties go out of scope, and the object references are released. This allows the dependent objects to terminate. If a dependent object has a Parent property, however, Visual Basic cannot not destroy the parent object in the first place, because the dependent object has a reference to it.

The dependent object cannot be destroyed, either, because the parent has a reference to it. This situation is illustrated for an out-of-process component in Figure 6.7.

Figure 6.7 Circular reference prevents objects from being destroyed.



19

Client application B has released its reference to its Widget object. The Widget object has a reference to a Knob object, whose Parent property refers back to the Widget object, keeping both objects from terminating.

A similar problem occurs if the Widget object contains a collection of Knob objects, instead of a single Knob. The Widget object keeps a reference to the Knobs collection object, which contains a reference to each Knob. The Parent property of each Knob

contains a reference to the Widget, forming a loop that keeps the Widget object, Knobs collection, and Knob object alive.

The objects client B was using will not be destroyed until the component closes. For example, if client A releases its Widget object, there will be no external references to the component. If the component does not have any forms loaded, and there is no code executing in any procedure, then the component will unload, and the Terminate events for all the objects will be executed. However, in the meantime, large numbers of orphaned objects may continue to exist, taking up memory.

Note If a circular reference exists between objects in two out-of-process components, the components will never terminate.

83

Circular References and In-Process Components

If you implement your component as a DLL, so that it runs in the process of the client application, it's even more important to avoid circular references. Because an in-process component shares the process space of the client application, there is no distinction between 'external' and 'internal' references to a public object. As long as there's a reference to an object provided by the component, it stays loaded.

This means that a circular reference keeps an in-process component loaded indefinitely, and the memory taken up by orphaned objects cannot be reclaimed until the client application closes.

For More Information Circular references and some techniques for dealing with them are demonstrated in the ObjModel.Vbg sample application.

84