Object-based Programming

Introduction

COM programming is based on an object-oriented style of programming. In VB, this means the use of classes and class-based object references. If you are comfortable defining your own classes with logical properties and methods, and then using these classes to instantiate and manipulate objects, you can safely skip this section and continue the tutorial here. Otherwise, let's begin!

We will study object-based programming in the context of an example: a form-based progress indicator class called CProgress. Here's a picture of this class in action (you should see this for yourself by running the application \VBCOM\Demos\Class-based Progress Indicator\App.exe; if you have not already done so, you can download the labs from the Setup page):

Note the naming convention --- classes always being with a C. It will be helpful if you also think of the C as standing for a concrete class, i.e. a class that can be turned into a run-time object that fulfills some function.

First, a bit of terminology. A class is compile-time entity that a programmers writes. In VB, a class is produced by adding a Class Module to your project. A class may contain zero or more members, where a member is either a property (think of it as a variable) or a method (subroutine or function). At run-time, you instantiate a class to produce an object that resides in memory. In fact, you can instantiate the same class over and over again, each time producing a distinct object. Finally, to facilitate discussion, we will use the terms client and server to denote two objects that are interacting, in particular where the client object is accessing a property or invoking a method on the server object:

Using the Class CProgress

The class is quite simple in design, having only one public property and two public methods:

```
Public Value As Integer '** percentage done
Public Sub Show() '** show the progress form
Public Sub Hide() '** hide the progress form
```

When this class is instantiated into an object, the Value of the object represents the current status of the progress indicator, e.g. 40 (denoting 40%). By changing this value, the client can show progress. The object's methods, Show and Hide, are invoked by the client to draw and erase the progress indicator form, respectively. Altogether, this list of public properties and methods denotes the interface between clients and CProgress servers.

At this point, let's look at some sample client code which instantiates and uses the CProgress class. If you haven't already, open the supplied VB project \VBCOM\Demos\Class-based Progress Indicator\App.vbp and view the code in the cmdProgress button of the main form:

```
Dim progress As CProgress '** object reference to
  '**  CProgress instance
Dim i As Integer, _
   j As Variant

Set progress = _
    New CProgress '** new instance of CProgress
```

The first line declares an object reference variable (a pointer in C++ parlance) of type CProgress. Initially, an object reference variable points to no object, and thus has the value Nothing. The third line uses VB's New operator to instantiate an object of class CProgress, and then sets the object reference variable progress to this new instance. At this point we have:

The code continues by displaying the progress indicator form, and then executing a loop which simulates a long-running operation:

```
progress.Show '** invoke Show method

For i = 1 To 10
  progress.Value = _
    progress.Value + 10 '** update by 10%
  For j = 1 To 1000000: Next j '** pause to simulate activity
Next i
```

Each time through the loop, progress increases by 10%. Finally, once the loop completes, the progress indicator form is erased from the screen and the progress variable reset to Nothing:

```
progress.Hide '** invoke Hide method
Set progress = Nothing '** explicitly destroy instance
```

This last step is not strictly necessary, since VB will reset the variable to Nothing automatically once it goes out of scope. However, it is important to note that the number of references to an object determines the lifetime of that object, i.e. the duration it exists in memory. By setting progress to Nothing we are explicitly reducing the reference count of that object by 1. Once an object's count reaches 0, it is destroyed. Thus, in the example above, the object is immediately destroyed after the Set statement is executed.

Implementing the Class CProgress

If you haven't done so already, open the supplied VB project \VBCOM\Demos\Class-based Progress Indicator\App.vbp and view the code in the Class Module CProgress (stored in the file CProgress.cls). Using VB's browser (F2), the class is summarized as follows:

The class actually contains two properties and four methods, although the first 3 listed above are private to the class (and thus accessible only to VB and the object itself). The private property frm is used by the object to hold a reference to the Form instance containing the progress indicator. This form is created in the method Class_Initialize, which is called automatically by VB whenever a new CProgress object is instantiated. Likewise, the form is destroyed in the method Class_Terminate, called automatically when the object's reference count reaches 0. Here's the relevant code from the class CProgress:

```
Option Explicit

Private frm As Form '** object reference to progress form

Private Sub Class_Initialize()
  Set frm = New frmProgress '** create a new instance of frmProgres
  Me.Value = 0 '** initialize our value to 0%
End Sub

Private Sub Class_Terminate()
  Unload frm '** destroy progress form instance
  Set frm = Nothing
End Sub
```

In this case, the method Class_Initialize is called behind the scenes in response to the client executing New:

```
Set progress = New CProgress '** create a new
  '**  instance of CProgress
```

Thus, when developing a class, use the Class_Initialize method to initialize the server for upcoming client use. As for Class_Terminate, in this case the method is called when the client clears the last reference to the server:

```
Set progress = Nothing '** explicitly destroy instance
```

Therefore, use the Class_Terminate method to cleanup (close files, destroy helper objects, etc.) before the server is gone for good.

Now for the public interface of CProgress: Value, Show and Hide. Firstly, the Value property is not implemented using an integer variable, but instead is a logical property physically realized by

appropriate Get and Let methods (look closely at the bottom pane of the browser window above). Logical properties offer many benefits: validation, the ability to take action when a property changes, read-only data, etc. However, a stronger motivation is that COM simply does not allow a class to contain public data properties. In a COM-compatible class, the only public members may be methods.

Ignoring COM for the moment, a logical Value property makes sense in this situation since the server needs to update the progress indicator in response to a change in value. The Let method is used to capture a write to the logical property, much like the Let statement in VBA assigns a new value to a variable. Note that the new value is passed as a parameter to the method:

```
Public Property Let Value(ByVal percentage As Integer)
  frm.pbarProgress.Value = percentage '** update progress bar
  frm.lblPercentage.Caption = _
    CStr(percentage) & "%" '** update label
  frm.Refresh '** redraw form
End Property
```

The new value is simply assigned to the underlying Windows 95 ProgressBar control on the progress form, as well as written to a label. In order for the client to be able to read the current progress value, a corresponding Get method is also provided:

```
Public Property Get Value() As Integer
 '** return current state of progress bar
  Value = frm.pbarProgress.Value
End Property
```

Get methods are actually functions which return the current value of the logical property.

The final two members of CProgress are the Show and Hide methods. These are perhaps the easiest of the bunch, since their function is obvious:

```
Public Sub Show()
  frm.Show '** show form and redraw
  frm.Refresh
End Sub

Public Sub Hide()
  frm.Hide '** erase form but leave form object in memory
End Sub
```

That's it, the class CProgress is now completely implemented!

Classes vs. Instances

Before we begin our discussion of COM, let's review the notion of classes, and instances of classes. Consider the following client code:

```
Dim p1 As CProgress, p2 As CProgress, p3 As CProgress

Set p1 = New CProgress
Set p2 = New CProgress
Set p3 = New CProgress

p1.Value = 40
p2.Value = 5
p3.Value = 99

p1.Show
p2.Show
```

```
p2.Hide

Stop '** pause client and activate the debugger
```

How many instances of CProgress exist in memory? Three! How many progress indicator forms exist in memory (i.e. instances of frmProgress)? Three! How many of these form instances are visible on the screen? One! Okay, now you're ready to start the first lesson on VB COM...

COM Classes and Servers

Introduction

In Visual Basic, building a COM-compatible object is deceptively easy: given the class definition for this object, you simply add the class to a particular type of VB project, make the project, and you're done. That's it, end of tutorial! In reality of course, there's more to VB COM than just doing make.

First, let's define some standard COM terminology. A coclass refers to a COM class, i.e. a class that is compatible with COM. What makes VB so powerful is that all classes created in VB are COM-compatible. It will be helpful if you also think of a coclass as denoting a concrete class, a class that contains functionality / code. A COM component is synonymous with coclass.

A COM server is one or more coclasses compiled into a single, distributable file. COM servers typically have the file extension .dll or .exe. A COM server must be registered on a client's machine before that client can instantiate any of the coclasses in the server.

There are two types of COM servers, in-process and out-of-process. If a client instantiates a coclass from an in-process COM server, the resulting object lives in the same address space (and process) as the client. However, if a client instantiates a coclass from an out-of-process COM server, the resulting object will live and run in a separate process --- on the same machine (local) or a different machine (remote). To build an in-process COM server in VB, you create an ActiveX DLL project, add one or more class modules, and perform make; the result is a file with the extension .dll. To build an out-of-process COM server, use an ActiveX EXE project type; make will then yield a file with the extension .exe.

In-Process VB COM Servers

The extension DLL stands for Dynamic Link Library, implying that an in-process COM server is linked into the client upon the first instantiation of any of its coclasses. Note that the entire COM server is linked in, not just the coclasses being instantiated. For example, suppose a COM server has 3 coclasses, and a client instantiates coclass1 and coclass2. Here's the resulting situation in memory on that machine:

Now suppose the user, on this same machine, starts up another process called Client #2 that instantiates coclass3. Conceptually, the situation in memory is now:

In essence, each process gets its own copy of COM Server. Why is this important? Because if COM Server contains "global" variables, it is important to note that these variables are not shared by the different client processes.

Out-of-Process VB COM Servers

As we all know, the extension EXE stands for executable program. Thus, out-of-process COM servers are completely separate processes, running asynchronously until a client makes a method call into one of the server's objects (at which point the client object blocks until the server object returns from the method call). For example, suppose we have an out-of-process COM server with 3 coclasses, and a client object on the same machine instantiates coclass2. The situation in memory is:

What if multiple clients instantiate coclasses from the same COM server? Typically, each instantiation yields a different object within the same server process:

However, whether these server objects run concurrently --- i.e. whether method calls from different clients execute at the same time --- depends on how the COM server is configured; we'll save this topic for later, when we discuss COM activation.

Note that the configuration of the COM server controls other things as well. In particular, whether or not: (1) the different server objects in the above picture can share "global" variables, and (2) instantiation yields multiple objects in the same server (as shown above) or an entirely new server process each time (thus # of processes = # of server objects).

LAB: Building VB COM Servers

Time for a lab exercise! The goal is to take an existing application, remove one of its classes, turn that class into a COM component, and produce a new, COM-based application. First we'll build an ActiveX DLL COM server, then repeat the exercise by building an ActiveX EXE COM server. The application simulates a long-running operation, displaying a progress indicator as it proceeds. Here's a picture of the app, which you can see this for yourself by running \VBCOM\Labs\COM Servers\App.exe (if you have not already done so, you can download the labs from the Setup page):

It is the progress indicator, defined by the class CProgress in \VBCOM\Labs\COM Servers\App.vbp, that will become a COM component.

Startup VB, and create a new ActiveX DLL project. VB will define an initial class module called Class1. Select this class, and remove it from the project (Project >> Remove); when prompted, do not save changes.
Now add the existing class module file "CProgress.cls" to the project: Project >> Add Class Module, Existing tab, navigate to \VBCOM\Labs\COM Servers\CProgress.cls, and open. Likewise, add the existing form "frmProgress.frm" to the project.
Take a moment and review the code in CProgress. For help understanding this code, jump here.

View the project explorer (View >> Project Explorer), expand the Class Modules, and select CProgress. View the properties window (F4), and change the Instancing property from Private to MultiUse. This makes the COM coclass publicly accessible to clients.
Modify the project's properties (a very important step when building a COM server) via Project >> Properties. Under the General tab, set the name to "ProgressDLL" and the description to "_Progress Indicator In-Process Server (VBCOM)". Click OK.
Finally, make your COM server via File >> Make, into the same directory as the other files. Save your work, and exit VB.

Congratulations, you have just built your first COM server! Your COM server resides in the file "ProgressDLL.dll" that you made; if you cannot see this file in your lab directory, make sure your explorer window is set to display hidden files. Now, to test your work, let's build a client application.

Startup VB, and create a new Standard EXE project. Remove the initial form module Form1, and add the existing form \VBCOM\Labs\COM Servers\frmMain.frm.
Modify the project's properties: set the startup object to be frmMain, and the name to "Client". Click OK.
View the form, in particular the code behind the cmdProgress command button. The sub creates an instance of CProgress, shows progress in increments of 10, and then destroys the instance:

```
  Dim i As Integer
Dim j As Variant
Dim progress As CProgress

  Set progress = New CProgress '** instantiate COM component
  progress.Show '** tell component to show itself

  For i = 1 To 10 '** update progress
    progress.Value = progress.Value + 10
    For j = 1 To 1000000: Next j '** pause to simulate operation
  Next i

  progress.Hide
  Set progress = Nothing '** destroy COM component
```

Now run the application (F5). Depending on how VB is configured on your machine, you will either get an immediate compiler error ("User-defined type not defined", with CProgress highlighted), or the app will startup and show the main form. In the latter case, clicking the button will then yield the same error message: "User-defined type not defined".
The problem is that our new client application is unable to locate the class CProgress. Whenever you want to use a COM component, you must first reference it so that VB can find it. Thus, view the list of registered COM components on your machine (Project >> References), and select the progress indicator component we built earlier --- i.e. check the item "_Progress Indicator In-Process Server (VBCOM)". Click OK.
Now re-run the application. It should work perfectly, merrily showing progress in increments of 10!
Finally, make an executable "Client.exe", save your work, and exit VB.
You should be able to run "Client.exe" outside of VB, and it should run identically to "App.exe". However, the former uses COM, and the later does not. If you want to run multiple clients and view multiple progress indicators on the screen, you will need to add a call to DoEvents inside the client's For-Next loop, and perhaps pause a bit longer as well.

At this point, it is very important to observe the following: no special programming was required to use COM with VB! The only differences from the original "App.exe" were configuration-oriented --- the use of a different project type in the server and setting a project reference in the client. On the surface, COM programming in VB is that easy.

For completeness, let's build the progress indicator into an out-of-process COM server. Once again, the only changes will be the use of a different project type for the server, and a different project reference in the client:

Repeat steps 1 .. 6 above, except create an ActiveX EXE project, set the project name "ProgressEXE" and the description to "_Progress Indicator Out-of-Process Server (VBCOM)". After you make your project, you should end up with a COM server in the file "ProgressEXE.exe".
Reopen the "Client.vbp" project, and under project references uncheck the in-process COM server and check your new "_Progress Indicator Out-of-Process Server (VBCOM)". Now run your client (F5), and it should function as before!
To convince yourself that the COM server actually runs as a separate executable, insert a Stop statement in the client immediately after the instantiation of CProgress:

```
Set progress = New CProgress '** instantiate COM component
Stop
progress.Show '** tell COM component to show itself
```

Now run the client from inside VB, and click the command button to execute the above code. VB should enter break mode with the Stop statement highlighted. Bring up the Task Manager on your computer, and you will see that ProgressEXE is running! Switch back to VB, continue execution, and let the client run (i.e. let the processing of cmdProgress's Click event run to completion). Return back to the Task Manager, and observe that ProgressEXE is no longer running. Why not? Because the client no longer has a reference to a CProgress object, and when the last object in a VB COM server is destroyed the server is stopped.
If you want to play... To keep the COM server running, we need to keep at least one object alive in the server. This means we need to keep the object's reference count > 0. One way to do this in our clients is to keep object references in global variables, which never go out of scope. In the client app, first move the declaration of the progress variable from the Click event to the Declarations section (top) of the main form. Then, instead of instantiating CProgress in the Click event, now do this in the form's Load event. Finally, move the "Set progress = Nothing " statement from the Click event to the form's Unload event. Now when you run the client, the COM server will be started as well, and this same process will continue to run until the client terminates. Confirm this behavior using the Task Manager. That's enough lab work for now, good job!

Registering a VB COM Server

As mentioned in the introduction, a COM server must be registered before a client can instantiate any of the coclasses in that server. However, in the above lab the client app (e.g. "Client.exe") ran just fine without any kind of explicit "registration" step. How can this be? Who registered the progress indicator COM server? And what does it mean for a COM component to be registered?

In short, registering a COM server means updating the Windows registry database on that computer. The registry enables COM to locate coclasses in response to client instantiation requests. To ease component development, VB automatically registers --- on your development machine --- any COM server you make (i.e. File >> Make). Thus, each time you do make, VB unregisters the previous version (if any) and registers the new version.

As evidence, let's break the "Client.exe" application by simply unregistering the COM server on which it depends. First, in preparation, locate the file \VBCOM\Misc\regsvr.reg and double-click to execute; this file adds some entries to your registry database that makes server registration and unregistration easier. Next, run your "Client.exe" in the lab directory \VBCOM\Labs\COM Servers\ and convince yourself that it shows progress without error. Now right-click on the file "ProgressDLL.dll" --- the in-process progress indicator COM server --- and select "Unregister COM Server" from the pop-up menu. Finally, re-run "Client.exe" and click the command button to show progress: this should yield "Run-time error 429: ActiveX component can't create object." In other words, the requested COM server is not registered on this machine. [Note: if you did not receive an error, perhaps your client is instantiating the out-of-process version of the progress indicator. Unregister "ProgressEXE.exe" and try again.] To fix the application, simply re-register the COM server by right-clicking on the appropriate file.

How can you find out what COM servers are registered on your machine? One way is to view project references. Startup VB, select Project >> References, and you should see something similar to:

This list is produced by VB using information culled from the machine's registry database; recall we used this dialog in the lab to configure the client.

COM servers can also be registered via the command-line. To register / unregister an in-process COM server, use the REGSVR32 utility:

```
REGSVR32 ProgressDLL.dll
REGSVR32 /u ProgressDLL.dll
```

In the case of out-of-process COM servers, run the server itself:

```
ProgressEXE.exe /RegServer
ProgressEXE.exe /UnregServer
```

These approaches make it easy to automate the registration process, in particular during application installation on a client machine. This task is typically performed by the install program itself.

For now, we can safely ignore the registration problem since VB will automatically register COM servers for us. However, you need to keep this in mind when it comes time for deployment. We will revisit this issue in more detail when we present COM activation.

In-Process vs. Out-of-Process?

When developing a COM component, you are faced with an immediate decision: create an in-process server, or an out-of-process server? The short answer is that in-process COM servers offer better performance while out-of-process COM servers offer greater flexibility. The former are more efficient since the server objects live in the same address space as the client, enabling a much cheaper communication mechanism (a simple subroutine call). The latter are more flexible since they can be deployed remotely on server machines, without having to recompile any of the code.

At first glance, it would appear that multi-tier applications are best designed using out-of-process COM servers. However, it is becoming quite common to develop in-process COM servers and then deploy

these components using surrogate processes. A surrogate process acts as a host for an in-process COM server, allowing it to run in an address space separate from the client --- just like an out-of-process COM server. However, the advantage is that the surrogate can provide services to its server objects, such as support for sharing state (i.e. "global variables"). The most important example of a surrogate process is Microsoft Transaction Server (MTS), which among other things provides support for distributed transactions. Thus, if you want to build COM components that work with MTS, you must create in-process COM servers.

What's next? A discussion of client-side COM programming.

Client-side COM Programming

VB programmers have long been client-side COM programmers: anyone who has programmed a control (e.g. the Text Box) has programmed a COM object. Thus, we are all familiar with manipulating an object's properties and calling an object's methods. However, there is more to being a competent client-side programmer. In particular, there are different ways to instantiate COM objects, different ways to declare your object reference variables, and important COM-related functions such as TypeOf. Finally, polymorphism is an important client-side programming technique that should be used whenever possible.

Note that there are two distinct styles of client-side COM programming, compiled vs. scripting. The former is typically done via a compile-time tool such as VB6, while the latter is often done using a text editor (Notepad?) for execution in a run-time environment such as IE. As we'll see below, the former offers better performance and ease of programming, while the latter offers flexibility. But as we'll also see later (in our discussion of server-side COM programming), these two styles will impact the design of our COM objects as well.

Instantiating COM Objects

There are two ways in VB to create an object from a COM coclass, New and CreateObject. New is merely an optimized form of CreateObject, either performing one less registry lookup or bypassing COM altogether (when the coclass resides in the same VB project as the instantiating class). To minimize the chance of naming conflicts when using New, always refer to a coclass using the format ServerName.ClassName. For example, given the progress indicator coclass we built in the previous COM Servers lab, here is the proper way to instantiate a progress indicator object with New:

```
Set progress = New ProgressDLL.CProgress
```

Note that when COM servers are built in VB, the server name is derived from the project name. A call to CreateObject looks very similar:

```
Set progress = CreateObject("ProgressDLL.CProgress")
```

In this case, however, the coclass is denoted by a string (known as a ProgID). Even though resolving the string requires an extra registry lookup, this approach provides important flexibility since the string can be generated at run-time (e.g. based on configuration information or user input). CreateObject will also accept an additional string parameter, denoting the computer on which to instantiate the object:

```
Set progress = _
  CreateObject("ProgressDLL.CProgress", "CompanyServer")
```

Of course, this only works if the remote computer is properly configured (more on this later in COM activation).

Declaring Object References

While you may not have given it much thought in the past, it turns out that how you declare your variables is very important with respect to COM. In general, you have two choices when declaring a reference to a COM object: use the generic type Object, or a specific class-based reference. For example, to declare a reference to a progress indicator object, either of the following will work:

```
Dim progress As ProgressDLL.CProgress  '** class-based reference
```

or

```
Dim progress As Object  '** generic object reference
```

The former declares that progress can only reference an instance of the CProgress coclass (a run-time error occurs otherwise); the latter allows progress to reference any type of object. The trade-off is performance and ease of programming versus flexibility.

When using class-based references, you are bound directly to the object via what is known as v-tablebinding, the most efficient form of binding available for COM objects. [The term v-table stands for virtual table, the technique used by C++ compilers for implementing virtual methods and dynamic binding.] Class-based references also enable IntelliSense and compile-time type checking, since the compiler knows exactly what type of object is being manipulated:

As you'll see in a moment, v-table binding is typically 3-10x faster than that available when using an Object-based reference (which relies upon what is known as late binding).

So why would anyone ever use a generic Object-based reference? The first reason is flexibility. Suppose you want to iterate across a collection of different types of objects, or write a subroutine that accepts different types of objects as a parameter. For example, consider the following subroutine InsertAtFront, which inserts a string into the first position of either a List Box or a Combo Box. In particular, note that the second parameter is of type Object, allowing the caller to pass a reference to either type of object:

```
Public Sub InsertAtFront(s As String, control as Object)
  control.AddItem s, 0
End Sub
```

Since both List Boxes and Combo Boxes understand the AddItem method, this subroutine executes fine --- unless of course the caller passes some other type of object, in which case a run-time error will occur ("Object doesn't support this property or method").

The second reason for using Object-based references is quite simple: it is the only reference type supported by scripting languages such as VBScript. For example, consider the following VBScript code, which instantiates and initializes a new progress indicator:

```
Dim progress
Set progress = CreateObject("ProgressDLL.CProgress")
progress.Value = 0
MsgBox "It worked!"
```

[You can run this code if you have WSH (Windows Scripting Host) installed: place the code in a text file, give the file a .vbs extension, and then double-click to execute. If it fails, make sure the ProgressDLL COM server is registered (forget how to register? see registering a COM server).] Note that scripting code is also limited to the use of CreateObject; the New operator is not supported (since class-based references in general are not supported).

LAB: Performance Trade-offs

As you'll see in an upcoming discussion of COM interfaces, the underlying difference between v-table binding and late binding is that the former binds at compile-time, while the latter binds entirely at run-time (hence the slower performance and lack of compile-time type-checking). Let's witness this difference firsthand.

Your first task is to create an in-process COM server and save your work in \VBCOM\Labs\Client-side\. If you have not already done so, you can download the labs from the Setup page. In particular:

1. Startup VB, and create a new ActiveX DLL project. VB will define an initial class module called Class1. Rename this class CWorker, and define a single public method called SomeTask as follows

```
Public Sub SomeTask(i As Long, s As String)
 '** nothing
End Sub
```

2. Make sure the the Instancing property of CWorker is set to MultiUse.
3. Modify the project's properties (a very important step when building a COM server) via Project >> Properties. Under the General tab, set the name to "Workers" and the description to "_Workers that don't really work (VBCOM)". Click OK.
4. Finally, make your COM server via File >> Make, producing "Workers.dll". Save your work, and exit VB.

Recall that a side-effect of making a COM server (via File >> Make) is that VB automatically registers the server on your development workstation. Now let's build a client application to demonstrate the differences in binding. As before, save your work in \VBCOM\Labs\Client-side\ as you proceed:

5. Startup VB, and create a new Standard EXE project. On Form1, place 2 command buttons and 2 labels as shown below (the second label has its BorderStyle property set to "Fixed Single" to make it appear like a text box -- name this label lblTime):

1. Reference the COM server you just built: Project >> References, check "_Workers that don't really work (VBCOM)".
2. Add the existing module "basClock.bas" to your project via Project >> Add Module; this file enables us to time things, and is located in the directory you are working in, \VBCOM\Labs\Client-side\.
3. Now let's program the Click event of the "V-table Binding" command button to time 100,000 calls to our COM server:

```
Const calls = 100000

Private Sub Command1_Click()
  Dim i As Long, worker As Workers.CWorker
  Set worker = New Workers.CWorker

  lblTime.Caption = ""
  lblTime.Refresh
  StartClock

  For i = 1 To calls
    worker.SomeTask i, "a string"
  Next i

  lblTime.Caption = StopClock
End Sub
```

9. Run the application (F5), click the button, and record the time.
10. Next, program the Click event of the "Late Binding" command button. The code is identical to the above code, except for the variable declaration of worker:

```
Private Sub Command2_Click()
  Dim i As Long, worker As Object
```

Convince yourself that IntelliSense does not work with Object-based references; you'll see this when you enter the following line of code as part of the For-Next loop:

```
  worker.SomeTask i, "a string"
```

11. Now re-run the application, click the second button, and record the time. How much slower is late binding?
12. Finally, make an executable "Client.exe", save your work, and exit VB. Then run "Client.exe" outside of VB and record the times. You should witness essentially the same performance difference.

If you want to experiment further, build an out-of-process COM server (be sure to use a different project name and description), and time calls to it from the client. You should find that out-of-process COM is another 10x slower.

Determining an Object's Type

Is it possible to get the flexibility of Object-based references, yet the performance and safety of class-based references? The answer is yes, with a little bit of work and RTTI (run-time type identification). VB's TypeOf function will query an object at run-time to determine if it's of the requested type. If so, you can then safely "type-cast" to a class-based reference. For example, recall the InsertAtFront subroutine presented earlier. Here is an updated version, rewritten using TypeOf to enable compile-time type-checking and more meaningful error handling:

```
Public Sub InsertAtFront(s As String, control as Object)
  If TypeOf control Is ListBox Then
    Dim lst as ListBox
    Set lst = control '** type-cast to a list box
    lst.AddItem s, 0 '** we know list boxes can AddItem
    Set lst = Nothing
ElseIf TypeOf control is ComboBox Then
    Dim cbo as ComboBox
    Set cbo = control '** type-cast to a combo box
    cbo.AddItem s, 0 '** we know combo boxes can AddItem
    Set cbo = Nothing
  Else
    Err.Raise <error code, source, "invalid parameter">
  End If
End Sub
```

Note that performance is probably not enhanced in this case, since the overhead of TypeOf and the type-cast (Set statement) will offset any advantage of v-table binding for the call to AddItem. However, if the object was being accessed more than once, a noticeable improvement in performance would result.

Polymorphism

When programming lots of similar objects, exploiting polymorphism is an important technique for writing client-side code that is resilient to change. An operation is polymorphic if it works on different data types. For example, the VB operator '+' is polymorphic since it operates on integers, reals, and strings. Classes are often designed with polymorphism in mind, allowing client-side programmers to write more general code that dynamically adapts as new classes are added to the system --- i.e. as the system evolves.

For example, suppose you are developing a GUI front-end for a sales application. Your company sells different types of products, e.g. Books, CDs, DVDs, Videos, etc. Each product type is thus represented by a class: CBook, CCD, CDVD, CVideo, etc. However, you recognize that these different classes share many features in common, for example every product has a ProductName. Since this property is polymorphic, we can manipulate it in a way that is independent of the actual products currently instantiated. For example, suppose products denotes a collection of product references. Then the following subroutine DisplayNames will display each product's name in a list box:

```
Public Sub DisplayNames(products as Collection, lst as ListBox)
  Dim product as Object

  lst.Clear

  For Each product in products
    lst.AddItem product.ProductName
  Next product
End Sub
```

Though less efficient (due to the lack of v-table binding), the advantage of this code is that it is product-independent: regardless of the types of products in the collection, the above subroutine will work without change. Thus, even as new product classes are added to the application, DisplayNames will not need to be rewritten.

Is polymorphism available via fast v-table binding? Yes, this is one of the many advantages of interface-based programming, which we discuss next.

Interface-based Programming

Introduction

When you manipulate an object, you do so via an interface: its public properties and methods. Objects must implement an interface to be useful to clients, and clients rely upon this interface to get work done. For example, the interface to our CProgress coclass (from the COM Servers lab) contains one property and two methods for manipulating, showing, and hiding a progress indicator:

```
Public Property Value As Integer
Public Sub Show()
Public Sub Hide()
```

This can be discovered by examining the coclass directly, or by referencing the COM server from a VB project and then using the object browser (F2). In general, an interface declares the name of each property/method, as well as types and parameter information.

The most important observation to make at this point is that an interface, once published and in use by clients, should not be changed. Doing so may break any number of existing --- and working! --- client applications. Thus, an interface should be viewed as a contract between an object and its clients.

The Default Interface

As a builder of objects, however, what happens if you need to change a class's interface? Perhaps you want to redesign an interface to be easier for clients to use. Or maybe it needs to evolve in a way you had not anticipated. Or more simply, perhaps the interface is just plain wrong and needs to be fixed. We know that changing an interface may break existing client code, yet leaving the interface as is forces new clients to work with an inadequate design. Is there a technique for evolving a class from one version to the next while remaining backward compatible with existing client applications?

The versioning problem stems from the fact that in traditional object-oriented programming, clients build dependencies directly on the class; in particular, on what is known as the default interface (i.e. the properties and methods marked public in the class itself). These dependencies prevent the class from evolving in a positive way. The solution is to separate a class's interface from its implementation, making the interface a first-class citizen that can be manipulated in its own right. By separating out the default interface, objects can now evolve by implementing additional interfaces. Existing clients continue to use the original interface, while new clients have a choice of using the original interface or a more recent one.

Given that object-oriented programming revolves around the notion of user-defined classes, the idea is to extend this to include user-defined interfaces as well.

User-defined Interfaces

A user-defined interface is a standalone class, listing a set of properties and methods but containing absolutely no implementation details. An interface is thus an abstract class, since it cannot be instantiated --- doing so would yield an object incapable of execution. Instead, one or more coclasses must implement the interface, providing an implementation which fulfills the contract.

For example, consider once again the progress indicator provided by the CProgress coclass. To redesign based on user-defined interfaces, the first step would be to create an abstract class named IProgress (typically in the same COM server) and declare the necessary properties and methods:

```
Public Value As Integer

Public Sub Show()
End Sub

Public Sub Hide()
End Sub
```

The second step is to modify the coclass CProgress, delete the default interface, and then implement IProgress. The coclass would now look something like:

```
Implements IProgress

Private Property Get IProgress_Value() As Integer
  <implementation>
End Property

Private Property Let IProgress_Value( _
    ByVal RHS As Integer)
  <implementation>
End Property

Private Sub IProgress_Show()
  <implementation>
End Property

Private Sub IProgress_Hide()
  <implementation>
End Property
```

Finally, the client-side code changes very little, except for variable declarations:

```
Dim progress As ProgressDLL.IProgress

Set progress = _
   New ProgressDLL.CProgress '** instantiate as before
progress.Show
. . .
```

Notice that the client now uses an interface-based reference to access the object (versus a class-based reference), thus breaking the dependence of the client directly on the coclass. Conceptually, the situation can be viewed as follows:

Each interface that an object implements appears as a separate "lollypop". Clients may interact with an object only through the interfaces symbolized by its lollypops.

Advantages?!

At this point you might be saying to yourself: "User-defined interfaces seem like a lot of extra work for no obvious benefit." While it is true that interface-based programming does require more effort, in the long run it is well worth the investment, for many reasons.

The first reason, as mentioned earlier, is to enable coclasses to evolve while remaining backward compatible with existing clients. This is accomplished by requiring that an object implement a new interface whenever it wants to make a change to an existing one. Suppose, for example, that we want our progress indicator to accept a real number for the indicator value instead of an integer. The solution is to define a second interface named IProgress2:

```
Public Value As Single

Public Sub Show()
End Sub

Public Sub Hide()
End Sub
```

and then implement both interfaces within the CProgress coclass:

```
Implements IProgress
Implements IProgress2
. . .
```

Existing clients continue to use IProgress, while new clients have a choice between IProgress or IProgress2:

The second reason motivating user-defined interfaces is encapsulation: all properties and methods of a coclass truly become private. The only public members of a class reside within the interfaces, and thus outside the coclass itself. The benefit is that clients are unable to see, and therefore depend upon, private implementation details. These details are then free to change over time.

The third reason is that efficient polymorphism is now available via v-table binding. For example, recall our earlier example of polymorphism involving a company selling different types of products: Books, CDs, DVDs, Videos, etc. Each product type is represented by a coclass (CBook, CCD, CDVD, CVideo, etc.), where these classes all share a common set of functionality (such as ProductName). Earlier, we programmed these objects polymorphically using an Object-based reference:

```
Dim product as Object '** an object-based reference

'** display each product within collection in a list box
lst.Clear
For Each product in products
  lst.AddItem product.ProductName
Next product
```

However, use of the Object data-type is inefficient since it relies upon late binding. The solution is to redesign your product classes with a user-define interface (IProduct), and then program against this interface in the client. Assuming that every product coclass implements IProduct, the resulting client code changes very slightly --- a different variable declaration. However, this trivial change is enough to enable v-table binding:

```
Dim product as _
   ProductServer.IProduct '** interface-based reference!

'** display each product within collection in a list box
lst.Clear
For Each product in products
  lst.AddItem product.ProductName
Next product
```

Conceptually, the objects within the collection now appear as follows:

The final reason for learning user-defined interfaces is that COM programming is based largely upon interface-based programming. In other words, a COM programmer is an interface programmer. Thus, to master COM, you must master interfaces.

LAB: Client-side Programming with User-defined Interfaces

In this lab we will focus on client-side interface programming. The goal is to work with interface-based references, polymorphism, and versioning. Note that in a later section we'll shift our attention to server-side interface programming, where you'll get a chance to build the COM server you'll be using here.

Ultimately you will build two clients, a version 1 and a version 2. This will simulate the evolution of a client-side app in response to the evolution of the underlying COM server. To orient yourself, explore the directory \VBCOM\Labs\Interfaces-client-side\ (if you have not already done so, you can download the labs from the Setup page). Notice there are two in-process COM servers, "ProductServer(v1).dll" and "ProductServer(v2).dll". Likewise, notice there are two client-side apps in the Solution\ sub-directory, "Client(v1).exe" and "Client(v2).exe". Due to user-defined interfaces and good programming techniques, either client app works with either COM server; you'll learn some of these techniques in the lab.

First, let's see how the client applications behave. [NOTE: although you can define new products via the client app, this product information is not stored and thus does not appear the next time you start the client.]

1. Register the v1 COM server; see registering a COM server if you forget how. This version declares an interface called IProduct, and three coclasses that implement it: CBook, CCD, and CVideo. Now run "Client(v1).exe" from the Solution\ sub-directory. Note that there are 3 types of products: books, CDs, and videos. The list box displays the name of each such product, and for each you can view 3 kinds of information: type, price, and quantity. Here's a screen snapshot:

Now run "Client(v2).exe", and observe that it behaves exactly the same, i.e. the new client is backward-compatible with v1 of the COM server. This is due to the fact that "Client(v2).exe" checks before using any features specific to v2 of the COM server.

2. Now unregister v1 of the COM server, and register v2. This COM server has an additional product coclass (CDVD), and an additional interface (IProductExtraInfo) that is implemented by two of the coclasses. For backward-compatibility, the coclasses also continue to implement the IProduct interface, enabling the COM server to support v1 clients. As proof, run v1 of the client app ("Client(v1).exe"). Note that it behaves exactly as before, with the exception that it displays DVDs as well. However, to see the impact of the new interface, run v2 of the client app ("Client(v2).exe") which is designed to explicitly take advantage of IProductExtraInfo whenever possible:

Notice that some books now display additional information, such as the book's author. Also, observe that the type of extra information may differ, e.g. some books have a "Pub Date" while others do not. This information is being provided via the IProductExtraInfo interface, which v2 of the client is smart

enough to use when implemented by an object. One other coclass also implements this interface; can you determine which it is? [Hint: what other type of product has extra info?]
Okay, let's build version 1 of the client.

3. First, unregister v2 of the COM server and re-register v1.
4. Now start VB by opening up the project "Client.vbp" in \VBCOM\Labs\Interfaces-client-side\. Reference the v1 COM server: check the item "_Product Server for Client-side interface-based programming." Bring up the browser (F2), and browse the ProductServer library. You should see 3 coclasses and one interface. Browse each of the coclasses --- note that no public members appear, since the default interface of each coclass is empty. In other words, all coclass details are private! The only way to manipulate these coclasses is via a user-defined interface, in this case IProduct. Unfortunately, VB's browser does not tell you which interfaces are implemented by which coclasses; you learn by way of documentation (or by using a tool such as OleView, which we'll introduce later). Familiarize yourself with the interface IProduct.
5. When the client app starts, it builds a collection of product objects, displays the available product types in the drop-down list box named cboTypes, and then triggers the Click event for cboTypes to load all the product names into the list box lstProducts. This is all done in the main form's Load event. You should pretend that the initial product information is being read from a database (even though it is actually being generated by hand); see the module basProducts for more details.
6. Your first task is to implement the private subroutine ListProducts behind the main form. This routine is called at various times to display the relevant product names in lstProducts. You will either display every product in the collection (which is referenced via a global variable called products), or just those products of a particular type (e.g. "Book"). This is controlled by the current selection in the drop-down list box, available via cboTypes.Text. Use a For-Each loop to iterate through the collection, and an object reference variable of type ProductServer.IProduct. Note that by programming against this interface, your code will remain compatible with future versions of the COM server.
7. When you are done, save your work and run the client app. You should be able to view product names in the list box; be sure to select each different product type from the drop-down list, and make sure 5 products are displayed in each case (there are 5 books, 5 CDs, and 5 videos in the "database").
8. The next step is to display a product's type, price and quantity in the grid grdInfo when its name is clicked upon in the list box. In particular, modify the Click event of lstProducts to first locate the appropriate product in the collection, and then display its information in rows 0, 1 and 2 of the grid. Once again, be sure to use an object reference variable of type ProductServer.IProduct. Save and test your work.
9. The final step is to allow users to define new products via the form frmNew. This form is displayed modally in response to the user selecting Products >> New... from the main form. Modify the Click event of the OK button (cmdOK) to create an object of the appropriate type, initialize the object, and add it to the collection with the product's name as its key. To remain compatible with future versions of the COM server --- which may introduce new coclasses --- use CreateObject to instantiate the product (instead of New). Dynamically create the ProgID that CreateObject needs by concatenating the string "ProductServer.C" with the requested product type.
10. Save, run and test. Your client application should be fully functional. When you are satisfied, build a v1 executable of the client as "Client(v1).exe".

Exit VB, and run "Client(v1).exe" to make sure it works. Now unregister v1 of the COM server and register v2. Since v2 continues to support the IProduct interface, your v1client app should function properly without change; rerun your "Client(v1).exe" to convince yourself this is true. Also, since the v1 client uses CreateObject to instantiate objects, note that it automatically adapts to the presence of a new coclass --- DVDs --- in the v2 product server.

Leaving v2 of the COM server registered, let's build a new version of the client app to take advantage of the server's new interface, IProductExtraInfo.

11. Reopen "Client.vbp" and confirm that v2 of the COM server is currently referenced (i.e. that "_Product Server (v2) for Client-side interface-based programming" is checked). Then, under the Make tab of the project's properties, change the conditional compilation argument to "version = 2". This allows the code in basProducts to properly instantiate the products at program startup when v2 of the COM server is used.

12. Browse the ProductServer library and study IProductExtraInfo. It consists of three methods that a coclass must implement as follows:

```
'** adds a (key, value) pair to the product
'**   (e.g. "Author", "S. King")
Public Sub AddInfo(key As String, value As String)

'** gets the ith (key, value) pair (1 <= i <= InfoCount)
'**   and returns it via parameters,
'**   also returning True if successful else False
Public Function GetInfo(i As Long, key As String, _
   value As String) As Boolean

'** returns the number of (key, value) pairs
'**   for this product
Public Function InfoCount() As Long
```

Your goal in the client is to use this interface --- if and only if it's available --- to display a product's extra info in the grid when the product's name is clicked upon in the list box.
Thus, modify the Click event of lstProducts as follows. First, use TypeOf to query the particular product being displayed in the grid to see if it implements ProductServer.IProductExtraInfo. If so, type-cast the existing product object reference (via a Set stmt) into another object reference variable of type ProductServer.IProductExtraInfo. Then, for each (key, value) pair the object contains, add another row to the grid (Rows = Rows + 1), display the key in column 0 of the new row, and display the value in column 1.
That's it. Now save, run, test. When you click on books and videos, you should see extra product information; CDs and DVDs do not currently support IProductExtraInfo. Finally, build a v2 of your client app into the file "Client(v2).exe" and exit VB.
Outside VB, run "Client(v2).exe" and make sure it works properly. Before we finish, it is worth emphasizing the subtle importance of using TypeOf to query an object at run-time (vs. assuming that an object supports a particular interface and just using it). For example, unregister v2 of the COM server and re-register v1 --- the use of TypeOf is why the v2 client app still runs correctly, even with an older version of the COM server that never heard of IProductExtraInfo! Likewise, suppose the coclasses CCD or CDVD (or some other coclass) implement IProductExtraInfo in a future version of the COM server --- the run-time call to TypeOf will then return True, and v2 of the client app will automatically display the extra product information!

As you can see, interface-based programming, combined with good programming techniques, can enable smooth evolution of software on both the client and the server.

What's Next?

Before we can discuss interface-based programming on the server-side, we need to lay the foundation for exactly how COM servers are located, and how COM objects are instantiated. This will introduce you to GUIDs, the registry, class factories, and the like. Hence our discussion of COM activation.

COM Activation

IDL

As you are now aware, a COM programmer is an interface programmer : programming against interfaces on the client-side, and implementing interfaces on the server-side. Conceptually, COM programmers view the world as follows:

Recall that one of the motivations for COM is its ability to support clients and servers written in different languages (hence the famed DevelopMentor battle cry that "COM is love!"). This ability is based upon a language-neutral approach for defining interfaces. Instead of C++ or Java or VB, an IDL (Interface Definition Language) is used to define each interface; in particular, COM uses MIDL (Microsoft IDL, pronounced "middle"). MIDL is often referred to as the language of COM.

For example, recall the IProductExtraInfo interface your programmed against in the previous lab. When viewed using VB's browser, this interface appeared as:

```
Public Sub AddInfo(ByVal key As String, ByVal val As String)
Public Function GetInfo(ByVal i As Long, key As String, _
   val As String) As Boolean
Public Function InfoCount() As Long
```

In reality, however, the interface was actually defined by the following MIDL:

```
interface IProductExtraInfo

  HRESULT AddInfo([in] BSTR key, [in] BSTR val);
  HRESULT GetInfo([in] long i,
            [in, out] BSTR* key,
            [in, out] BSTR* val,
            [out, retval] VARIANT_BOOL* );
  HRESULT InfoCount([out, retval] long* );
};
```

The above MIDL was compiled into the COM server ("ProductServer(v2).dll") when it was built. This interface information was then accessed by VB when you referenced the server via Project >> References, thereby enabling browsing, IntelliSense, type-checking and v-table binding. As we'll see in a moment, compiled interface definitions will be needed for other reasons as well.

While a detailed discussion of MIDL is beyond the scope of this tutorial, a few things should be readily apparent from studying the previous MIDL:

All COM methods are functions returning an HRESULT (32-bit integer).
VB types are converted into equivalent MIDL types.
ByVal parameters are passed in to the server, while ByRef parameters are based in and back out.
VB functions are internally defined with an extra return (i.e. out) parameter.
Although most VB COM programmers know very little about MIDL, the more you learn, the easier it is to understand COM and communicate with other COM programmers.

Type Libraries and GUIDs

Most COM-based tools and services do not parse MIDL directly. Instead, they expect MIDL in its compiled form --- a type library. Every COM server typically has an associated type library, either compiled into the server or as a separate file (with a .tlb extension). For example, when you reference a COM server in VB (i.e. Project >> References), VB looks for and reads the type library. Likewise, when you build a COM server using VB (i.e. File >> Make), by default VB writes a type library into the COM server file for you.

Exactly what information does a type library contain? Firstly, it declares the existence of all public interfaces and coclasses in the server. For example, in the previous lab, this is how VB knew that v2 of ProductServer contained two user-defined interfaces (IProduct and IProductExtraInfo) and four coclasses (CBook, CCD, CDVD and CVideo). Secondly, to eliminate the chance of name-collision with other COM servers, type libraries define the mapping from text-based names to globally unique identifiers (128-bit integers). Known as GUIDs (pronounced "ga-wids"), every interface, coclass, and the type library itself is assigned a different GUID. GUIDs are conveyed using MIDL's uuid keyword (universally unique id), written in hex (yielding 32 hex digits), and unique throughout the world.

Type libraries primarily contain these two sets of information: a list of all public interfaces and classes, and their GUIDs. Thus, looking at v2 of ProductServer in more detail, we see initially the names of the type library as well as the user-defined interfaces:

```
[ uuid(D2A5D8EC-B8B1-11D3-9EA9-0010A4F15889) ]
library ProductServer
{
  [ uuid(D2A5D8ED-B8B1-11D3-9EA9-0010A4F15889) ]
  interface IProduct
  {
    ...
  };

  [ uuid(D7411E76-B94C-11D3-9EAB-0010A4F15889) ]
  interface IProductExtraInfo
  {
    ...
  };
```

Interface GUIDs are known as IIDs (interface ids). Then comes each coclass. Note that VB-generated coclasses always contain a default interface. In ProductServer, since none of the coclasses contain public properties/methods, all such default interfaces are thus empty:

```
[   uuid(D2A5D8EF-B8B1-11D3-9EA9-0010A4F15889) ]
  interface CBook { }
;

  [ uuid(D2A5D8F0-B8B1-11D3-9EA9-0010A4F15889) ]
  coclass CBook {
    [default] interface CBook;
    interface IProduct;
    interface IProductExtraInfo;
  };

[ uuid(D2A5D8F1-B8B1-11D3-9EA9-0010A4F15889) ]
  interface CCD { };

  [ uuid(D2A5D8F2-B8B1-11D3-9EA9-0010A4F15889) ]
  coclass CCD {
    [default] interface CCD;
    interface IProduct;
  };
```

```
uuid(D7411E78-B94C-11D3-9EAB-0010A4F15889) ]
  interface CDVD { };

  [ uuid(D7411E79-B94C-11D3-9EAB-0010A4F15889) ]
  coclass CDVD {
    [default] interface CDVD;
    interface IProduct;
  };

[ uuid(D2A5D8F3-B8B1-11D3-9EA9-0010A4F15889) ]
  interface CVideo { };

  [ uuid(D2A5D8F4-B8B1-11D3-9EA9-0010A4F15889) ]
  coclass CVideo {
    [default] interface CVideo;
    interface IProduct;
    interface IProductExtraInfo;
  };
};
```

Coclass GUIDs are known as CLSIDs (class ids). Later, when we discuss server-side COM programming, you'll see how to view the actual type library.

Here a GUID, There a GUID

Even though VB hides all traces of GUIDs from both client-side and server-side COM programmers, you still must be aware of them. In particular, you need to know that when you build a v-table bound client application, the GUIDs from the referenced type library are compiled into the app. For example, the following client-side COM code

```
Dim product As ProductServer.IProduct '** interface-based object reference
Set product = New ProductServer.CBook   '** instantiate CBook coclass

If TypeOf product Is ProductServer.IProductExtraInfo Then
  ...
End If
```

would be compiled to reference coclass D2A5D8F0-B8B1-11D3-9EA9-0010A4F15889, as well as interfaces D2A5D8ED-B8B1-11D3-9EA9-0010A4F15889 and D7411E76-B94C-11D3-9EAB-0010A4F15889. Even late-bound clients, which use a generic object-based reference, are compiled to a specific IID:

```
Dim product2 As Object   '** generic object-based reference
```

(more on this when we discuss COM interfaces). On the other hand, note that using CreateObject (instead of New) results in a different dependency --- on the ProgID instead of the underlying CLSID:

```
Set product = CreateObject("ProductServer.CBook")
```

In this case only the string is compiled into the application.

When building a COM server, the same sort of thing happens there as well. Each coclass is compiled with the GUIDs of the interfaces it implements, and the server itself is compiled with the GUIDs of the coclasses that can be instantiated.

Finally, whenever you reference a COM server (i.e. its type library) in a VB project, you are embedding into that project a reference to the type library's GUID.

At this point, you may be starting to wonder how all these GUIDs get mapped to the proper entity: "If a VB project references type library D2A5D8EC-B8B1-11D3-9EA9-0010A4F15889, how is the actual type library found?" Or: "How is a reference to CLSID D2A5D8F0-B8B1-11D3-9EA9-0010A4F15889 in the client turned into an instance of CBook on the server?" The latter is an especially good question when you consider that the server could be anywhere on the network...

The Registry

Now we have come to the main thread of this discussion: COM activation. Exactly how are COM servers and their type libraries located? Once located, how is a coclass instantiated (activated)? How does activation impact future method calls between client and server? Type libraries, GUIDs, and the registry all play a role in answering these questions.

Recall that before a COM server can be used, it must be registered on the client machine. For example, an in-process COM server can be registered using the REGSVR32 utility:

```
REGSVR32 ProductServer(v2).dll
```

When you register a COM server, you are really just copying information from the server's type library into the machine's registry database. In particular, ProgIDs are added to section HKEY_CLASSES_ROOT (HCR), CLSIDs to section HCR\CLSID, IIDs to section HCR\Interface, and type library GUIDs to section HCR\TypeLib. Since these entries are all inter-related, they typically refer to each other as well: ProgIDs reference CLSIDs, CLSIDs and IIDs reference the type library, etc.

To see what actually gets stored in the registry, go ahead and register v2 of ProductServer (it's safe to do this regardless of which version of the server is already registered). Then run regedit, open HCR, and look for the ProgID "ProductServer.CBook". Open this key, and you should see the underlying

CLSID that it maps to:

Now look for the CLSID key under HCR, and locate CLSID D2A5D8F0-B8B1-11D3-9EA9-0010A4F15889. Open this key, and you'll find:

In particular, notice the sub-key entitled "InprocServer32". This identifies the coclass as part of an in-process COM server, where the default value for this sub-key declares the location of the server file:

With enough patience, you should be able to locate each ProgID, CLSID, IID, and the type library within the registry. While you may not yet understand each of the sub-keys you encounter, you will shortly.

COM Activation

Activation is the process of creating an actual object in memory from a client-side call to instantiate --- regardless of where the coclass may live. COM's built-in support for this location transparency is one of its most important motivations.

There are three distinct activation cases: in-process, local, and remote. Obviously, the first case involves in-process COM servers; the latter two both involve out-of-process COM servers. [For a review of the differences and trade-offs, click here.] Keep in mind that in each case, the client-side instantiation code remains the same, for example:

```
Set product = New ProductServer.CBook
```

What determines the particular activation case is (a) the configuration information in the registry on the client and server machines, and (b) the type of COM server. And if the COM server is built using VB, changing its type is simply a matter of modifying a project property, not the coclasses themselves.

In each case, the same basic activation sequence shown below is followed (assume for now that the client is using VB's New operator to initiate activation). The call to New first triggers a call from the VB run-time to the COM run-time (signified by "OLE32.dll"); the CLSID and IID are passed from VB to COM at this point. COM's Service Control Manager then gets involved (signified by "RPCSS.exe", and affectionately known as the "SCuM"), and is responsible for locating the server via registry lookup

of the CLSID. If the server is not already loaded/running, the SCuM will load the DLL or start the EXE running, as needed. Next the SCuM calls into the server's Class Factory object, asking it to instantiate the given coclass (based on the CLSID) and return a reference to the desired interface (based on the IID). Class factories are responsible for the actual creation of COM objects at run-time, and every COM server must contain a class factory object capable of instantiating each coclass in the server. Once the object has been created, the class factory object returns the appropriate interface-based reference back to the SCuM, which in turn returns the reference back to VB and the client app. At this point the activation is complete, COM's work is done, and now the client can begin making method calls on the server.

What if the client uses CreateObject instead of New? VB's run-time simply performs an initial registry lookup to map the given ProgID to the needed CLSID. Activation then proceeds as discussed above.

In-process Activation

In the in-process case, activation follows the above sequence exactly. You can view a PowerPoint animation of the process, clicking the mouse to advance. Note that the animation instantiates two objects, yet the DLL is loaded only once (and into the same address space as the client app). Also, keep in mind that once the activation process is complete, the client has a direct reference to the object --- the SCuM is no longer involved.

Local Activation

In the case of local activation, the server runs out-of-process but on the same machine as the client. The most important difference from the in-process case is that additional objects are required to implement the remote procedure call (RPC) communication mechanism used with out-of-process COM servers. COM uses RPC because it mimics standard procedure calling, an approach already familiar to programmers. These additional objects, known as the proxy and stub, create the illusion that the client

is calling the server directly. In reality, however, the client calls the proxy, the proxy communicates with the stub, and the stub calls the server:

The proxy and stub work together to enforce the semantics of a standard procedure call in the remote case --- across process (and potentially machine) boundaries. Thus, they perform two critical functions: (1) blocking the client until the server returns from the call, and (2) marshalling parameters back and forth between client and server.

Note that the client and server are oblivious: the proxy impersonates the server, and the stub impersonates the client. In fact, for all intents and purposes, the proxy is the server from the perspective of the client; likewise, the stub is the client from the perspective of the server. This is how COM achieves location transparency, since the proxy and stub are inserted at run-time as needed. On the other hand, COM is not transparent with regards to performance: method calls to local, out-of-process COM servers are at least 10x slower than equivalent in-process servers. This is primarily due to the overhead imposed by the proxy and stub. [You may even have witnessed this slowdown yourself in an earlier lab experiment; if not, you are encouraged to go back and try it when you are done here.]

Here is a PowerPoint animation of the local activation sequence. Observe how the proxy and stub are inserted at run-time, in particular with the proxy in the client and the stub in the server. But where do the proxy and stub come from? Interestingly, while it is possible to write your own proxy-stub code (in C++), it is much more common for a custom proxy-stub pair to be generated at run-time based on the interface being accessed by the client. When the SCuM needs a proxy-stub, it does a lookup of the IID in the registry to locate both (a) the type library that defines this interface, and (b) the CLSID denoting the proxy-stub code. For example, here's what you'll find in the registry for IID D7411E76-B94C-11D3-9EAB-0010A4F15889 (IProductExtraInfo) assuming v2 of ProductServer:

By default, COM servers produced using VB rely upon COM's Universal Marshaler (in "OLEAUT32.dll") to build custom proxy-stub code at run-time based on type library information.

Since proxy and stub objects are based on the target interface, note the implication: proxy-stub code is generated and inserted on both a per-object and a per-interface basis. For example, the following client code instantiates an object with two interfaces and then sets a variable to reference each one:

```
Dim product As ProductServer.IProduct, _
    extra As ProductServer.IProductExtraInfo
Set product = New ProductServer.CBook
Set extra = product
```

The result is two proxy-stub pairs: the first from product to the object's IProduct interface, and the second from extra to the object's IProductExtraInfo interface.

Remote Activation

The final case is that of remote activation, in which the server runs out-of-process on a different machine than the client. The primary difference between this and local activation is that the client's SCuM has to communicate with a remote SCuM in order to activate the desired coclass on the server. Otherwise the process is identical. The following PowerPoint animation summarizes all three activation sequences.

Note that in the remote case, the COM server must be registered on both the client and the server machines. In addition, the server's type library must be installed on the client machine to support proxy generation; obviously, the COM server and its type library must be installed on the server machine. Finally, CLSIDs on the client and server may refer to optional registry information in the form of an AppID (COM Application ID, under HCR\AppID). On the client, an AppID contains the name of the remote server machine to be used for activating the coclass. This value represents the default, however, and can be overridden if the client instantiates by providing a second parameter --- the machine name --- to CreateObject:

```
Set product = CreateObject( _
    "ProductServer.CBook", "<remote machine name>")
```

In this case, the SCuM ignores the AppID registry setting (if any) and tries to activate the coclass on the specified machine. On the server-side, an AppID is used to override the default settings for such things as access, launch control, and strength of authentication/privacy. Typically, the utility *dcomcnfg* is used to configure AppIDs on the server machine, although this is done automatically if you are using **Microsoft Transaction Server** (MTS).

Observations...

Interfaces are more than just a programming style --- they enable COM to achieve language-independence and location transparency. And while COM activation is a potentially expensive process, these goals are generally worth the cost.

What about COM *deactivation*? In other words, how are objects destroyed? COM uses a technique known as **reference counting**, in which each object maintains an internal count of how many references are currently pointing to it. When another reference is set, the client is responsible for informing the object; likewise when a reference is cleared. Eventually, when its count reaches zero, the object destroys itself. The good news is that VB automates the process of deactivation; you'll see how when we discuss COM interfaces.

But first, it's time to consider COM programming from the perspective of the server.

Server-side COM Programming

Interfaces

At this point we have discussed three important aspects of COM: interfaces, client-side programming, and activation:

Now it's time to focus on the fourth and final piece --- the server.

Server-side COM programming amounts to implementing one or more interfaces in a public coclass. Recall that there are two types of interfaces, the default and user-defined. The default interface contains all public properties and methods in the coclass; every coclass defined using VB has a default interface, even if it's empty. A coclass may also implement any number of user-defined interfaces.

For example, let's revisit the progress indicator COM server.

The server contains one coclass CProgress, which has a very simple default interface:

```
Public Value As Integer '** percentage done
Public Sub Show() '** show the progress form
Public Sub Hide() '** hide the progress form
```

How is this interface implemented? As shown in \VBCOM\Labs\COM Servers\Solution\ProgressDLL.vbp, CProgress ("CProgress.cls") simply defines the following four public methods (if you have not already done so, you can download the labs from the Setup page) :

```
'** hidden reference to underlying progress form
Private frm As Form

Public Sub Show()
  frm.Show
  frm.Refresh
End Sub

Public Sub Hide()
  frm.Hide
End Sub

'** get percentage done
```

```
Public Property Get Value() As Integer
  Value = frm.pbarProgress.Value
End Property

'** update percentage done
Public Property Let Value( ByVal percentage As Integer)
  frm.pbarProgress.Value = percentage
  frm.lblPercentage.Caption = CStr(percentage) & "%"
  frm.Refresh
End Property
```

Note that the Value property is implemented as a logical property with Get and Let methods: this is required by the rules of COM, since data properties are now allowed (interfaces may contain methods only). Also, note that the coclass's Initialize and Terminate events are used to create and destroy, respectively, the underlying progress form referenced by frm.

What about user-defined interfaces? Once defined, these can be implemented by any number of coclasses using VB's Implements keyword. For example, suppose we want to evolve the progress indicator with a Cancel button so that the user can cancel the operation being performed. First we define an appropriate interface, IProgressCancel:

```
Public Sub EnableCancel() '** show the Cancel button
'** returns True if button was pressed
Public Function Cancelled() As Boolean
```

This interface allows the client to show a Cancel button, and then detect if in fact the button was ever pressed. Next, we extend the CProgress coclass to implement the interface:

```
Implements IProgressCancel

Private Sub IProgressCancel_EnableCancel()
  frm.cmdCancel.Visible = True
  frm.Refresh
End Sub

Private Function IProgressCancel_Cancelled() As Boolean
  IProgressCancel_Cancelled = frm.Cancelled
End Function
```

Notice the methods are private --- they are only accessible through the interface. Finally, the underlying progress form is modified to include a hidden command button cmdCancel that when clicked, sets the form's Cancelled flag to True.

LAB: Implementing IProgressCancel

Since interfaces are so important, let's take a moment and implement IProgressCancel. Then we'll modify the client to check our implementation. You'll be working in the directory \VBCOM\Labs\Server-side\; you should run the client app ("Client.exe") in the Solution\ sub-directory before continuing (don't forget to register "ProgressDLL.dll" first). Here are screen snapshots of what you'll see:

The first step is to implement the interface in the CProgress coclass.

1. Close the Solution\ sub-dir, and startup VB by opening the project \VBCOM\Labs\Server-side\ProgressDLL.vbp. Notice that IProgressCancel has already been defined, and that frmProgress already contains a hidden command button (cmdCancel) and a public boolean property (Cancelled).
2. Open CProgress, and just below Option Explicit type "Implements IProgressCancel".
3. Using the wizard bar (the two drop-down lists just under the code window's title bar), select IProgressCancel from the left drop-down list. Using the right drop-down list, stub out each method in the interface by selecting it. Note that you must stub out every method, otherwise VB will not compile your coclass; recall that an interface is a contract which you must fulfill completely.
4. Implement each method as shown earlier.
5. Now build your COM server via File >> Make. Save your changes and exit VB.
6. Now let's rewrite the client and test our COM server's new functionality.

7. Startup VB by opening the project \VBCOM\Labs\Server-side\Client.vbp. Run the client and convince yourself that the new COM server is backward-compatible with the existing client (which has no notion of the new interface nor the Cancel button).
8. In cmdProgress's Click event, declare another object reference of type IProgressCancel. Then, if the "Enable Cancel button" check box is checked and the progress indicator object supports this interface (TypeOf), type-cast the existing reference to the new type (Set) and enable the Cancel button.
9. Inside the loop, if the Cancel button has been enabled then exit the For-Next loop if the Cancel button has been pressed. Note that it is the client's responsibility to call DoEvents during each iteration of the loop to allow the recognition of other events (such as the pressing of the Cancel button).
10. Finally, clear all object references at the end of the sub (always good practice when you're done with a reference).
11. Run your client app, and test all 3 cases: Cancel button not enabled, Cancel button enabled but never pressed, and Cancel button enabled and pressed. When you are satisfied, build an executable client app and test it outside of VB.

12. That's it for now; a more involved server-side lab is coming shortly. Good work!

Mapping VB to COM

As we motivated earlier, MIDL is the true language of COM. You can learn a lot about how VB and COM work by studying the MIDL that VB generates when you build a COM server. The easiest way to do this is using OLEView, a tool that is installed as part of Visual Studio (typically with VC++). If you have OLEView installed (by default it appears in Microsoft Visual Studio Tools), run it, scroll to the bottom where it says "Type Libraries" , and open to reveal the list of all type libraries registered on your machine. Find the one for the COM server you just built in the previous lab --- "_Progress Indicator v2 In-Process Server (VBCOM)" --- and double-click to reveal the MIDL. OLEView is reverse engineering the MIDL from the type library. [Note: if you do not have OLEView installed, you can still view the MIDL by opening the text file "ProgressIndicator.idl" in the directory \VBCOM\Demos\.]

What appears in the type library is every public coclass and interface defined in the COM server; Bas and Form modules are considered private and thus never shown. Also, observe that every VB class is mapped to a coclass with a default interface (and 0 or more other interfaces). As you would expect, the coclass CProgress is defined with 2 interfaces:

```
[
  uuid(A47EF721-C625-11D3-9ECD-0010A4F15889),
  version(1.0)
]
coclass CProgress {
  [default] interface _CProgress;
  interface _IProgressCancel;
};
```

Likewise, a user-defined interface in VB --- really just an abstract class --- is mapped to a noncreatable coclass with a default interface:

```
[
  odl,
  uuid(A47EF71E-C625-11D3-9ECD-0010A4F15889),
  version(1.0),
  hidden,
  dual,
  nonextensible,
  oleautomation
]
interface _IProgressCancel : IDispatch {
  HRESULT EnableCancel();
  HRESULT Cancelled([out, retval] VARIANT_BOOL* );
};

[
  uuid(A47EF71F-C625-11D3-9ECD-0010A4F15889),
  version(1.0),
  noncreatable
]
coclass IProgressCancel {
  [default] interface _IProgressCancel;
};
```

Finally, notice some of the attribute keywords: "oleautomation" tells COM to use the Universal Marshaler, while "hidden" prevents this information from appearing in VB's browser or IntelliSense.

To convince yourself that VB is actually reading the type library, note for example that it should be possible to declare object variables of type _IProgressCancel (the real name of the interface according to the MIDL) even though this type isn't available via IntelliSense. This is done as follows:

```
Dim cancel as [_IProgressCancel]
```

(the [ ] are needed since identifiers in VB cannot start with "_"). This declaration is equivalent to declaring the variable of type IProgressCancel, since its default interface is also _IProgressCancel.

Error Handling

Recall that every method in a COM-compatible interface is a function that returns an HResult (a 32-bit integer). You can see this by reviewing any COM server's MIDL:

```
interface _IProgressCancel : IDispatch {
  HRESULT EnableCancel();
  HRESULT Cancelled([out, retval] VARIANT_BOOL* );
};
```

COM uses negative HResults to denote error conditions. However, as a VB programmer you should never see raw HResult values: the VB run-time maps well-known COM error codes to VB-specific error codes and raises an exception (e.g. error #429, "ActiveX component can't create object" means that the underlying COM activation failed for some reason). Likewise, even though COM has its own internal exception handling mechanism, the VB run-time will map COM exceptions raised in the server into VB exceptions on the client.

Thus, unless there are good reasons to the contrary, your servers should raise exceptions to signify error conditions. In terms of error codes, raise either VB-specific codes or your own custom ones. The latter are best defined using an enumerated type:

```
Public Enum ServerErrorCodes
 '** a unique error code
  errInvalidArgument = vbObjectError + 512
  errUnknownKey ' +1
  errOutOfRange ' +2
  ...
End Enum
```

Note that vbObjectError is a predefined constant in VB denoting a range of available error codes; by convention, add 512 to the starting value. To raise one of these errors from your server back to the client, simply refer to it by name:

```
Err.Raise ServerErrorCodes.errInvalidArgument, _
    "<description>", ...
```

The client should trap errors as usual:

```
On Error Goto Handler

.
.
.

Handler:
  Select Case Err.Number
    Case ServerName.ServerErrorCodes.errInvalidArgument
```

```
    ...
  Case ServerName.ServerErrorCodes.errUnknownKey
    ...
  Case ServerName.ServerErrorCodes.errOutOfRange
    ...
  Case Else
    ...
End Select
```

In order for client-side programmers to have access to the enumerated values, they must be part of the server's type library; this occurs automatically if the Enum statement resides in one of your server's public classes.

Note that if you ever see a large, negative error code such as "-2147023174" when working with COM, you are most likely looking at a raw HResult that VB was unable to map into something more meaningful. This is either an unexpected error, or an untrapped server-side error that has made its way back to the client. At this point, you must look at the bits themselves to determine the source and type of error (in particular the Facility Code, bits 16-27, and the Error Number, bits 0-15). This implies that a good server-side programming practice is to trap all possible errors in the server, and raise only well-documented error codes back to the client.

Marshalling

In COM, clients and servers communicate by copying data back and forth in the form of parameters. A ByVal parameter results in a one-way transmission from client to server; a ByRef parameters results in a two-way transmission (from client to server and back again). Function return values are a special case, and lead to a one-way transmission from the server back to the client. As mentioned earlier, this information is conveyed in MIDL using the attributes in and out. For example, this VB function

```
Public Function SomeTask( _
    ByVal i As Integer, s As String) As Boolean
```

would appear in MIDL as:

```
HRESULT SomeTask([in] short, [in, out] BSTR*, [out, retval] VARIANT_BOOL*);
```

Why is this important? Because in the case of in-process COM servers, ByRef parameters are more efficient since ByVal requires data copying to enforce the semantics. However, in the case of out-of-process COM servers, ByVal is more efficient since the marshalling cost is only one-way.

In terms of data types, any VB type can be marshaled except for a fixed-length string. This is pretty impressive if you stop and think about it: million-character strings, multi-dimensional arrays, variants, user-defined types (UDTs). For example, a server-side method could return a dynamic array of user-defined records:

```
Public Type DataRecord
  Name As String
  MailCode As Integer
  ...
End Type

Public Function GetData( _
   ByVal filename As String) As DataRecord()
  Dim data() As DataRecord

  ... '** open file, redim array based on
  ... '**   amount of data, fill array
```

```
  GetData = data '** return a copy of the array
End Function
```

The client can then simply call this method to get a copy of the records:

```
Dim records() As ServerName.DataRecord, i As Long

records = server.GetData("<filename>")
For i = LBound(records) To UBound(records)
  <process records(i)>
Next i
```

COM will take care of marshalling the array from the server back to the client --- regardless of array size and server location. Note that the UDT must be defined in the server's type library; like Enum, this occurs automatically if the Type statement resides in one of your server's public classes. The use of UDTs also requires NT SP4 or later.

There is one very important case which demands additional discussion: reference parameters. Suppose a server-side method returns a Collection object:

```
Public Function GetCollection() As Collection

  Dim c As Collection

  Set c = New Collection '** instantiate and
  ... '**  populate the collection

  Set GetCollection = c '** return ?
End Function
```

What exactly is returned back to the client? A reference to the Collection object, not the object itself!

In particular, given an out-of-process COM server, upon return the situation is as follows:

The Collection object remains on the server, while the client ends up with a reference back to it (through a proxy-stub pair). This has obvious (negative!) performance implications when the client tries to manipulate the collection. The moral of the story is that in all but a few cases, objects themselves are never marshaled, but instead a reference to the object. One notable exception is that of ADO RecordSet objects, which are in fact marshaled between client and server.

Building a COM Server

After designing your interfaces and implementing your coclasses, building the actual COM server is the easy part: set a few project properties, and File >> Make. VB then builds a complete server including coclasses, type library, class factory, and self-registration code.

Before running File >> Make, however, your first decision is in-process vs. out-of-process. Keep in mind the trade-offs of performance vs. remote execution and fault tolerance. Also, consider surrogates like MTS which provide a value-added framework for hosting your in-process servers on remote machines. For in-process servers, leave the "Threading Model" set to "Apartment Threaded" (the default, see Project >> Properties). For out-of-process servers, you can create:

1. a single-threaded server ("Thread Pool" of 1), in which client requests are queued and executed one-by-one,
2. a limited multi-threaded server ("Thread Pool" of N > 1), in which at most N different objects within the server can be executing concurrently, or
3. a multi-threaded server ("Thread per Object") in which every object created by a client outside the server can be executing concurrently.

Note that MTS 2.0 is equivalent to case 2 where N = 100. More importantly, take note that the VB run-time (and COM!) work together to guarantee that a single VB object never executes two client requests at the same time, regardless of the caller --- a VB object always executes in a single-threaded manner, one request after another. This is achieved by executing VB objects in what are called a Single-Threaded Apartments (or STAs). Concurrency is achieved (safely!) by having multiple STAs within the same process; case 1 above has exactly 1 STA, case 2 has N STAs, and case 3 has an infinite supply. For example, suppose an out-of-process COM server was built with a "Thread Pool" of N = 2, and currently has 4 clients connected to 3 server objects as follows:

Since there are only two STAs, then either all 3 server objects live in the same STA, or there are 2 in one and 1 in the other (the situation above). In this case, then at most two server objects will execute concurrently: object #1 or object #2, and object #3. And if object #3 is executing, then it is running on behalf of either client #3 or client #4, but not both. At present, VB objects are assigned to an STA at activation, and you cannot influence this assignment; the object remains in the same STA until deactivated.

Next, before you build your COM server, think carefully about the values you choose for the project's Name and Description (Project >> Properties). The former becomes the programmatic name of the COM server, while the latter serves as the description seen by other VB programmers when they reference your COM server (i.e. select Project >> References).

If you have the Enterprise Edition of VB, consider checking the option "Remote Server Files" under the Component tab of Project >> Properties. This will generate a separate standalone type library, as well as a .VBR file that can be used with the VB utility Clireg32 for configuring client machines. Recall that when a client attempts remote activation, the client's registry must be properly configured and the appropriate type library must be installed.

Finally, always perform one last check of the Instancing property for each of your classes. The typical settings are Private, PublicNotCreatable, SingleUse, or MultiUse. Obviously, your interfaces should be PublicNotCreatable, and your public coclasses MultiUse. The SingleUse option is only available in out-of-process COM servers: a coclass set to SingleUse causes a separate instance of the COM server to be loaded and run each time this coclass is activated. Generally, you should avoid the use of GlobalMultiUse and GlobalSingleUse.

Versioning

We have come to one of the most important --- and subtle --- issues concerning server-side COM programming. The problem itself is easy to explain: suppose version 1 of your COM server has been deployed, and you need to release a new version (1.1) consisting of bug fixes. How do you ensure that your new COM server is backward-compatible with existing clients? Then, sometime later, suppose you want to release version 2.0 with added functionality. In this case you need to remain backward-compatible with existing clients, yet allow the creation of new clients that can exploit the added functionality. How? This problem is known as the versioning problem.

Actually, the versioning problem is not hard to solve, as long as you are careful in your design and maintenance of the COM server. In other words, it's less of a programming issue and more about project management.

The problem boils down to this: given an existing COM server, what can you safely change without breaking existing client code? The answer stems from what dependencies are built into those clients. First and foremost, clients depend on the interfaces to your coclasses. Hence your interfaces cannot change. In particular, if a coclass C implements an interface I, then C must continue to implement I. Furthermore, methods cannot be deleted from I, methods names cannot change, and method signatures must remain the same (i.e. parameter and return types). Obviously, clients also depend on the semantics of your methods (the function that each method performs), so this cannot change as well.

The more subtle dependencies arise from v-table bound clients. Recall that when a VB project references a COM server's type library, it build a dependency against the type library's GUID. Then, when the client is compiled against the type library, VB embeds the necessary CLSIDs and IIDs:

```
'** embed equivalent IID
Dim ref As ServerName.InterfaceName
'** embed equivalent CLSID
Set ref = New ServerName.CoclassName
```

Thus, v-table bound clients are dependent upon the COM server's GUIDs. On the other hand, late-bound clients (e.g. code written in scripting languages such as VBScript) have no such dependencies. This is due to the fact that late-bound clients use generic object-based references, and also ProgIDs instead of CLSIDs. Here's an example in VB:

```
Dim ref As Object
Set ref = CreateObject("ServerName.CoclassName")
```

As long as the client's machine is configured properly, late-bound clients will function correctly regardless of the server's GUIDs. Therefore, at least in the case of v-table bound clients, we see that your GUIDs cannot change.

In summary, there are two things that cannot change from one version of a COM server to another: its interfaces, and its GUIDs. The first one is easy to deal with --- never change an existing interface! Instead, define a new interface and have your coclasses implement this new interface in addition to the current ones. The second one is harder to deal with, since VB controls when a GUID changes via the "Version Compatibility" project property:

If you select "No Compatibility", then VB changes all GUIDs --- type library, CLSIDs and IIDs --- associated with this COM server each time it is recompiled. If you choose "Project Compatibility", then VB changes only the IIDs; the type library GUID and all CLSIDs remain the same. Finally, "Binary Compatibility" means that VB retains the same GUIDs as those in the reference version (a separate COM server file usually kept in a Release\ sub-directory, e.g. "Release\ProgressDLL.dll" in the screen snapshot above). In this case VB will also check to ensure that you haven't changed any of your interfaces, warning you during recompilation if you have.

Obviously, once you deploy the first version of your COM server, you should be in Binary Compatibility mode to ensure that future versions of your server remain backward-compatible. The purpose of Project Compatibility is to support development mode: the server-side programmer is free to change their interfaces, while client-side references to the COM server (and its coclasses) remain valid. Recall that type library references are stored in VB project files, while coclass references (CLSIDs) are often embedded in web pages. The last mode, No Compatibility, should be used in cases when the new version is to be completely incompatible with the previous versions --- in other words, start-over mode.

This should sound pretty straightforward. Develop your server in Project Compatibility, build a reference version in a Release\ directory, and then switch into Binary Compatibility mode. Unfortunately, there are a couple of potholes you need to be aware of. First, it is perfectly legal to add an interface / coclass to a COM server while in Binary Compatibility mode. However, since this entity does not exist in the reference version, VB will change its GUID each time you recompile. Thus, once you have finished development of your new interface / coclass, you must build another reference version for deployment. Over time, you will end up with numerous reference versions across

Release1\, Release2\, Release3\, etc. Note that your COM server will typically reference to the most recent version.

Second, VB considers it perfectly legal to add a method to an existing interface while in Binary Compatibility mode. As a budding COM programmer this should surprise you, since interfaces are supposed to be immutable! Thus, while a COM purist thinks in terms of identical vs. incompatible interfaces, VB has three possibilities: identical, compatible, or incompatible. Obviously, an interface is incompatible with a previous version if some method has changed, or been deleted altogether. An interface is compatible if it contains all previous methods (unchanged), yet adds one or more new ones. This is a compromise that allows an interface to evolve while remaining backward-compatible. However, to make this work, VB must internally maintain two separate IIDs --- one that denotes the previous interface, and another that denotes the new interface (previous methods + new methods). Existing clients continue to use the former IID, while new clients compiled against the new interface embed the latter IID. Eventually, when you are ready to deploy the new server, you must build another reference version, at which time both IIDs become fixed. Then, if you add yet again more methods, VB will internally maintain 3 IIDs, and so forth.

If compatible interfaces sound like a hack, you're right. In most cases they should be avoided, especially since they only work with default interfaces. The bottom line is this: if you are in Binary Compatibility mode, VB will not warn you if you add a method to an interface or a coclass (i.e. its default interface). This is okay in the latter case, since VB will correctly generate a compatible interface and maintain the necessary IIDs. However, in the former case --- adding a method to a user-defined interface --- VB fails to warn you but also fails to generate a compatible interface. Your new server no longer works with existing clients.

LAB: Server-side Programming with User-defined Interfaces

Recall the client-side front-end you built in an earlier lab, in which the underlying COM server (ProductServer) was given to you. In fact, you built two versions of the front-end, mirroring the

evolution of the COM server. Here is version 1:

Version 2 was written to exploit new functionality available in the server, in particular an interface for accessing extra product information:

In this lab you will develop the underlying COM server, of which there will be two versions. As a refresher, you may want to run the applications "Client(v1).exe" and "Client(v2).exe" in \VBCOM\Labs\Interfaces-client-side\Solution\ before continuing.

All lab files are available in the directory \VBCOM\Labs\Interfaces-server-side\. You will start with a fully-functioning ProductServer, version 1. The first step is thus deployment:

1. Startup VB by opening the server project "MyProductServer.vbp". View project properties, and note the server's name and description. View the Component tab, and note that the server is currently in development mode ("Project Compatibility"). Close the property dialog.
2. Before we can deploy, we must switch into Binary Compatibility mode. But before we can do that, we need to build a reference version. So, run File >> Make, create a new folder called "Release1", and build the DLL into this folder.
3. Now once again view the Component tab of project properties, and switch into deployment mode ("Binary Compatibility"). Make sure the reference version is set to "Release1\MyProductServer.dll", and click OK.
4. Rebuild your COM server, except this time save the DLL in the folder above Release1\ --- i.e. build the current version into the directory \VBCOM\Labs\Interfaces-server-side\.
5. Save your work and exit VB.
6. Now it's time to build v1 of the Client app and make sure MyProductServer works properly:

7. Change into the sub-directory "Client v1", and start VB by opening "Client.vbp". The client app is fully-functional, so all you have to do is build it. Note that "Client(v1).exe" is built into the directory above, i.e. where the server files live.
8. Save any changes and exit VB.
9. Run v1 of the client app and make sure you have Books, CDs and Videos. Note that there are no DVD products, nor is extra information available about products such as Books.

10. Pretend a few months have passed, and it's time to generate a new version of your COM server. The company now sells DVDs, and has also added extra product information to its database for Books and Videos. So let's develop version 2 of MyProductServer:

11. Reopen the server project "MyProductServer.vbp". Note that you will remain in Binary Compatibility mode.
12. Add a new class module, and name it CDVD. Make sure its Instancing property is set to MultiUse.
13. Implement the interface IProduct in coclass CDVD. Note that the other coclasses already implement this interface, so you might look there for help (or use Clipboard Inheritance).
14. Build the server, saving the DLL on top of the current version --- i.e. save into \VBCOM\Labs\Interfaces-server-side\, and not into the Release1\ sub-directory. You must leave the reference version alone.

15. Hide VB and run the client app; it should still work fine. In fact, you should also see and be able to create DVDs. This is due not only to Binary Compatibility, but also because the client app is smart and doesn't limit itself to a predefined set of CLSIDs. Instead, the client builds ProgIDs based on run-time information and then uses CreateObject to instantiate.

Back in VB, add another class module and name it IProductExtraInfo. Set the Instancing property to PublicNotCreatable. Define the interface as follows:

```
'** adds a (key, value) pair to the product
'**    (e.g. "Author", "S. King")
Public Sub AddInfo( _
    ByVal key As String, ByVal value As String)
End Sub

'** gets the ith (key, value) pair (1 <= i <= InfoCount)
'**   and returns it via parameters,
'**   also returning True if successful else False
Public Function GetInfo( _
  ByVal i As Long, ByRef key As String, _
  ByRef value As String) As Boolean
End Function

'** returns the number of (key, value)
'**  pairs for this product
Public Function InfoCount() As Long
End Function
```

1. Take a moment and save your work.
2. Now implement this interface in the coclass CBook. [Hint: define two private variables, one an empty array for holding the (key, value) pairs, and the other a long for storing the number of current pairs. In AddInfo, simply ReDim the array, store the data, and increase the count. The other methods should be self-explanatory.]
3. Save your work.
4. In order to test your implementation of the new interface, we need v2 of the client app (which has code written against this interface). You have three choices in how you run the client. First, you can create a project group consisting of the client and the server, and then just run the client from within VB; this is a great way to debug in-process VB COM servers, since you can set breakpoints in either project and step back and forth. The second approach is to make the server, and then test by opening the v2 client project and running from within VB. You can rebuild the server as needed, as long as you close and reopen the client project (this is necessary to successfully rebuild the DLL, since the client project is referencing the server's type library). The third approach is to build another reference version of the server (fixing CDVD's CLSID and IProductExtraInfo's IID), build v2 of the client into an EXE, and then you can safely run the EXE while rebuilding the server as needed.

5. Let's take approach #3, since the client is already written and now is as good a time as any to build the next reference version (since we don't plan to add other interfaces nor coclasses to this release of MyProductServer).

6. Back in the server project, run File >> Make, create a new folder called "Release2", and build the DLL into this folder.
7. Very important: change the project properties so that the reference version is now "Release2\MyProductServer.dll". Also, since we're here, change the Major version number to 2 (under Make tab).
8. Now rebuild the server again, except save the DLL above the Release2 directory. This is now your current version.
9. Save your work.
10. With that done, we can now build v2 of the client app without fear of breaking compatibility as we apply bug fixes to the server. Change into the sub-directory "Client v2", open "Client.vbp", and build the client app "Client(v2).exe" into the directory above. Close the client project.
11. Outside of VB, run the v2 client app. You should be able to view Books, CDs, DVDs and Videos. Also, if the interface was implemented properly, you should be able to click on a book and see

extra product information. If not, you need to debug the server (MsgBox debugging?), rebuild, and rerun the client app.

12. Once the server is working, the last step is to implement IProductExtraInfo within the CVideo coclass. Since this is simply an implementation detail, you can rebuild MyProductServer to make this added functionality available to the client app (i.e. there is no need for another reference version at this point).

13. Go ahead and complete the server by implementing IProductExtraInfo within CVideo. Rebuild the current version, save your work, and exit all running instances of VB.
14. The first test is to unregister the current version of MyProductServer and run both versions of the client app; neither should display any products.
15. Next, register Release1 of the server and run both versions of the client app; they should run identically, displaying Books, CDs and Videos, yet no extra information.
16. Unregister Release1 and register Release2. Both versions of the client app should run, but in this case differently.
17. Finally, unregister Release2 and re-register the current version. In this case v1 of the client should behave as before, but v2 should display extra product info for Videos as well.
18. That's it, excellent work!

Debugging

Unfortunately, debugging is a fact of life. The good news is that you can use VB's source-level debugger on both client-side and server-side VB code. You can set breakpoints, step back and forth between client and server, etc., with very few restrictions.

For in-process VB COM servers, simply create a project group in VB containing the client project and the server project(s). Run the client as you would normally, and debug. For out-of-process VB COM servers, keep the client and server as separate projects. The first step is to open the server project, set breakpoints, and start it running. Then open the client, set breakpoints, and start debugging. When you're done, don't forget to stop and restart the server as necessary. Note that out-of-process servers become single-threaded when debugged in this manner.

Limitations

Normally, public data in a .BAS module is global and thus shared throughout the project. However, in the case of COM servers, this is not necessarily the case. In fact, the only reliable way to share state amongst objects in a COM server is to build a single-threaded version, an expensive price to pay for global variables. But if you really need to, build your COM server with a Threading Model of either "Single Threaded" (for in-process) or "Thread Pool of 1" (for out-of-process). Or better yet, consider the use of a surrogate like MTS, which provides facilities for just this purpose.

Perhaps the most critical limitation is that scripting clients --- most commonly associated with web pages --- cannot access user-defined interfaces. In other words, scripting clients are limited to the default interface only. How significant is this? Very, given the importance of interface-based programming and the popularity of scripting clients. For example, consider the COM server you just finished building in the lab, MyProductServer. It contains 4 coclasses, none of which contain public methods --- hence their default interfaces are empty. Thus, MyProductServer is entirely unusable from the perspective of a scripting client (such as VBScript, JavaScript, WSH, IE, or ASP).

As a result, there are numerous (inelegant!) workarounds to enable scripting clients to access user-defined interfaces. Here are two of the most commonly used approaches. The first idea is to duplicate every interface-based method with a public method. For example, consider method InfoCount of interface IProductExtraInfo from the lab:

```
Public Function InfoCount() As Long
End Function
```

When implemented within a coclass, it appears as:

```
Private Function IProductExtraInfo_InfoCount() As Long
  ...
End Function
```

In this case, we simply add a public function that calls the private one:

```
Public Function InfoCount() As Long
  InfoCount = IProductExtraInfo_InfoCount()
End Function
```

This works fine assuming name collisions can be avoided and the interfaces are not changing (although you can always add methods). The second approach is a bit more work, but enables default interfaces to evolve in conjunction with user-defined interfaces. The idea is to wrap a custom coclass and its default interface around each user-defined interface:

Thus, a COM server with N user-defined interfaces needs at least N additional wrapper coclasses within the server. While there are various ways to implement wrapper classes, the following approach requires exactly N wrappers, regardless of the number of coclasses in the server. The example below is based once again on IProductExtraInfo, and would denote a new coclass CIProductExtraInfo within MyProductServer:

```
'** we have access to the user-defined interface
Private server As IProductExtraInfo

'** scripting client must call this first
Public Sub Connect(progid As String)
  Set server = CreateObject(progid)
End Sub

Public Sub AddInfo(ByVal key As String, ByVal value As String)
  server.AddInfo key, value
End Sub

Public Function GetInfo( _
   ByVal i As Long, ByRef key As String, _
   ByRef value As String) As Boolean
  GetInfo = server.GetInfo(i, key, value)
End Function

Public Function InfoCount() As Long
  InfoCount = server.InfoCount()
End Function

'** scripting client should call this last
```

```
  Public Sub Disconnect()
    Set server = Nothing
  End Sub
```

The code in the scripting client ends up looking very similar to v-table bound code:

```
Dim server As Object
  Set server = CreateObject( _
     "MyProductServer.CIProductExtraInfo")
  server.Connect "MyProductServer.CBook"
  server.AddInfo "Publisher", "MS Press"
  ...
  server.Disconnect
  Set server = Nothing
```

That's it.

Just a bit more COM...

If you've made it this far, congratulations are in order! You are well on your way to becoming a VB COM programmer. There is just one section remaining, since we have yet to discuss the two most important COM interfaces: IUnknown and IDispatch.

COM Interfaces

Introduction

No treatment of COM would be complete without a discussion of **IUnknown** and **IDispatch**, the two most important interfaces in COM. Every COM object is required to implement IUnknown, as it forms the underpinnings of *reference counting* and *RTTI* (run-time type identification). The IDispatch interface is what enables late-bound clients to call a COM object without any compile-time information.

IUnknown

The IUnknown interface consists of only 3 methods:

```
[
odl,
uuid(00000000-0000-0000-C000-000000000046),
hidden
]
interface IUnknown {
HRESULT QueryInterface( [in] GUID *riid, [out] void **ppv);
ULONG AddRef();
ULONG Release();
}
```

The first method, QueryInterface, supports RTTI with regards to interfaces: a client can call this method to see if the object supports a particular interface. The latter two methods, AddRef and Release, enable reference counting: a client calls AddRef to increase the object's reference count, and Release to decrease the count.

One of the rules of COM is that every coclass must implement IUnknown. In fact, every interface must include these three methods. Using MIDL, this is easily expressed via C++ style *inheritance*:

```
interface ISomeInterface : IUnknown
...
}
```

If you define your interfaces in VB, IUnknown is automatically inherited for you. More importantly, if you build your COM servers in VB, support for IUnknown is automatically implemented by each coclass.

The question is: **as a VB programmer, when do you call these methods**? Never! In fact, VB prevents you from calling IUnknown directly. Instead, the VB run-time calls the methods for you as appropriate. For example, in VB we use the TypeOf operator to see if an object implements a given interface:

```
'** call to underlying QI
If TypeOf product Is IProductExtraInfo Then
...
End If
```

This translates into a call to QueryInterface, where a reference to IProductExtraInfo's GUID is passed as the in parameter. Likewise, we use the Set statement to set and clear references to an object. These are translated into calls to either AddRef (and QI) or Release:

```
If TypeOf product Is IProductExtraInfo Then
Dim extra As ProductServer.IProductExtraInfo

'** 1. call to QI, and then AddRef if QI successful
Set extra = product

...

'** 2. call to Release
Set extra = Nothing
End If
```

Note that in case 1, the product object is first queried to see if it supports IProductExtraInfo (extra's data type); if the answer is yes, then AddRef is called since an additional reference will now exist to the object. Case 2 represents an explicit call to Release; if you forget to set a reference variable back to Nothing, VB is supposed to call Release for you when the variable goes out of scope.

Conceptually, support for IUnknown is denoted by a lollypop on top of an object:

In reality, when a v-table bound client connects to an object via some interface, the client is connected to the *virtual table* for that interface. Since every interface must support IUnknown, its methods are the first three entries in this table. For example, suppose a client is bound to the user-defined interface IProductExtraInfo. Then their object reference is really pointing to the underlying v-table:

A v-table is nothing more than a jump table to the actual methods.

IDispatch

Unlike IUnknown, which must be implemented by all COM objects, IDispatch is required only if the object wishes to support late-bound (i.e. scripting) clients. Objects which implement IDispatch are known as *scriptable* objects.

Why do late-bound clients require objects with a special interface? Recall that late-binding occurs in VB when we use the Object data type:

```
'** generic object-based reference
Dim obj As Object
'** instantiate some coclass...
Set obj = CreateObject(progid)
'** call some method...
obj.MethodA(...)
```

Given that COM programming is interface programming, the question becomes: "what interface is obj referencing?" Logically, it's the default interface, since that is the only interface available to late-bound clients. But the default interface is not known until the coclass is activated at run-time --- at compile-time, when VB has to generate code (e.g. to call MethodA above), it has no idea which coclass will be instantiated. Thus, the solution is for VB to assume the existence of a single physical interface in all scriptable objects, and that interface is IDispatch.

This implies that IDispatch must be very flexible, since this one interface must support all possible default interfaces. Interestingly, IDispatch inherits from IUnknown and then adds only four methods:

```
[
odl,
uuid(00020400-0000-0000-C000-000000000046),
restricted
]
interface IDispatch : IUnknown {
HRESULT GetTypeInfoCount(...);
HRESULT GetTypeInfo(...);

HRESULT GetIDsOfNames([in] GUID* riid, [in] char** rgszNames,
[in] UINT cNames, [in] ULONG lcid, [out] long* rgdispid);
```

HRESULT **Invoke**([in] long dispidMember, [in] GUID* riid,
[in] ULONG lcid, [in] USHORT wFlags, [in] DISPPARAMS* pdispparams,
[out] VARIANT* pvarResult, [out] EXCEPINFO* pexcepinfo,
[out] UINT* puArgErr);
};

In fact only the last two methods are strictly necessary. The first, GetIDsOfNames, maps a method name to a dispatch id. The second, Invoke, takes the dispatch id (and any in parameters), makes the call to the actual method for you, collects the results from the out parameters, and returns.

Once again, VB prevents you from calling this interface directly, and instead calls the methods for you. For example, consider the following late-bound code, which is accessing the default interface of the progress indicator coclass developed earlier:

```
'** interface-based reference to IDispatch
Dim progress As Object

'** Activate, QI, AddRef
Set progress = CreateObject("ProgressDLL.CProgress")

'** (1) get dispID for Value, (2) invoke with 0
progress.Value = 0
'** (1) get dispID for Show, (2) invoke
progress.Show

...

'** call to Release
Set progress = Nothing
```

In essence, every method call from the client through the default interface results in 2 calls through IDispatch. This, plus the overhead of processing the parameters, is the reason late-bound clients are often 3-10x slower than v-table bound clients *to the same object*. [We saw this slowdown in a previous lab.]

While conceptually a late-bound client is connected to the object's default interface, in reality it is v-table bound to the corresponding IDispatch interface. For example, in the case of the progress

indicator's default interface, the client really sees the following:

Observe how the default interface is available *behind* Invoke.

Dual Interfaces

As you might expect, if you define your interfaces in VB, they will automatically inherit from IDispatch. You may even have noticed this during our discussion of server-side COM programming,

when we used *OLEView* to reverse engineer MIDL from VB-generated type libraries. For example, here's the actual MIDL for the progress indicator's default interface:

```
[
odl,
uuid(A47EF720-C625-11D3-9ECD-0010A4F15889),
version(1.0),
hidden,
dual,
nonextensible,
oleautomation
]
interface _CProgress : IDispatch {
[propget] HRESULT Value([out, retval] short* );
[propput] HRESULT Value([in] short );
HRESULT Show();
HRESULT Hide();
};
```

Not only does it inherit from IDispatch, but notice the keyword attribute **dual**. A dual interface is one that can be used by both v-table bound and late-bound clients. VB always produces dual interfaces, and also generates the necessary support code when these interfaces are implemented by VB coclasses.
In other words, if you define your interfaces and build your COM servers strictly in VB, then your coclasses will automatically support both types of clients. VB accomplishes this by building virtual tables which form a union of IUnknown, IDispatch, and the interface's methods. Thus, in the case of the progress indicator's default interface, a client actually connects to the following v-table:

V-table bound clients have access to all 11 methods, while late-bound clients see only the first 7.
There is, however, a large caveat you must keep in mind: even though user-defined interfaces in VB are duals (and implemented as such), **late-bound clients are still limited to default interfaces only**. This limitation is orthogonal to the issue of dual interfaces.

Conclusions

We'll leave that to the next, and final, section. For good or bad :-), we're coming to the end of the tutorial...

Conclusions

COM

At the start of this tutorial, we motivated COM by stating its primary benefits: component-based development, language-independence, location transparency, and software evolution. As this tutorial comes to a close, we hope you have begun to see the importance of these benefits, and the effort to which COM goes in realizing them.

These benefits do not come cheap. As happens all too often in Computer Science, gains in programmability and maintainability come at the expense of performance. COM is no different. As a reference point, consider the following performance figures (all values represent "method calls completed per second"):

	in-process	**out-of-process (local)**	**out-of-process (LAN)**	**out-of-process (WAN, coast-to-coast)**
4-byte param	129,041	1765	495	18
100-byte param	119,240	968	342	16

While these figures are a few years old and may have changed, the implications have not: (1) out-of-process COM is 100-1000 times more costly than traditional method calls, and (2) cost is based primarily on the distance between client and server, not speed nor the amount of data. And there's not much you can do to reduce this cost, except minimize the total number of calls which occur between client and server.

COM has other costs as well, in particular administrative and training. The former requires good management, and the latter of course requires good training --- we hope this tutorial serves as a good start :-)

COM is expensive. Is it worth it? Yes.

VB COM

The good news is that Visual Basic significantly lowers the entry costs of using COM. As you have seen in this tutorial, VB and COM work together to automatically handle a staggering number of issues. From the client's perspective, VB:

- turns type libraries into IntelliSense
- takes care of the activation sequence
- can exploit both v-table and late binding
- generates the necessary calls to IUnknown
- handles differences in threading models
- marshals nearly all possible data types

On the server-side, VB automatically maps *any* class you define into a COM-compatible entity, be it an interface or a coclass. You can build an in-process or an out-of-process server with the flip of a property, and VB generates all the necessary infrastructure: type library, class factory, self-registration code, dual interfaces, threading support, marshalling, etc. And while VB's notion of binary compatibility is not perfect, it's a huge step in the right direction.

Still not convinced as to the power of VB? Then just ask any C++ programmer :-)

Cheers

Anyway, we hope you had fun, and learned a thing or two about COM. And perhaps, when your journey is farther along, you too will feel compelled to stand up in a crowded bar and yell "COM is love!".

Before you leave, a few serious recommendations. Look into surrogates for hosting your in-process COM servers, most notably **Microsoft Transaction Server** (MTS). And consider defining your interfaces separate from your coclasses, to ease the pain of maintaining binary compatibility; use MIDL or VB.

Take care!