Chapter 2

Designing, Building, and Working with COM-Based Components

Microsoft's Component Object Model is an important technology for sharing class-based code. The beauty of COM is that it is language independent, allowing developers to work in the language of their choice. It was VB, however, that first opened the door to widespread COM development.

The design of COM centers around the concept of an *interface:* Classes expose interfaces, and clients communicate with objects via these interfaces. Although VB can hide most aspects of interface-based programming, it's far better to be informed and to decide for yourself how much VB hides—and how much you explicitly embrace. If you are new to interfaces, rule 2-1 will get you started. We then encourage you to embrace fully interface-based design (rules 2-2 and 2-4), and to do so using tools outside VB (rule 2-3). Once

defined, an interface is considered immutable to maintain compatibility as the component evolves. Compatibility is a subtle issue, and is the subject of rule 2-5.

The remaining rules focus on other important but less traveled techniques with respect to COM: proper COM activation and termination (rules 2-6 and 2-7), high-level class design (rule 2-8), and the move away from ActiveX EXE servers (rule 2-9).

Note that you may come across some COM-related terms that aren't defined in great detail: *in-process* DLL, *GUID*s, *registering* a server, COM *activation,* and everyone's favorite *IUnknown.* Some of the rules assume a basic COM background, so readers new to COM may need to consult one of the many available COM texts. Or, you can review the free online tutorial designed for VB programmers at www.develop.com/tutorials/vbcom.

Rule 2-1: Think in Terms of Interfaces

An *interface* defines a communication protocol between a class and a client (a user of that class). When a client references an object, the interface associated with this reference dictates what the client can and cannot do. Conceptually, we depict this relationship as shown in Figure 2.1. Note that an interface is represented by a small "lollipop" attached to the object. This symbolizes the fact that an interface is separate from, but a conduit to, the underlying implementation.

But what exactly is an interface? Consider the following employee class `CEmployee`:

```
'** CEmployee: class
Private sName As String
Private cSalary As Currency

Public Property Get Name() As String
   Name = sName
End Sub
Public Property Get Salary() As Currency
   Salary = cSalary
End Salary
```

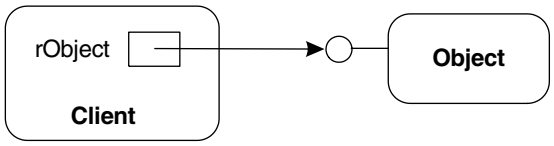


Figure 2.1 Client accessing an object through an interface

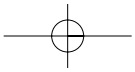
```
Public Sub ReadFromDB()
   ...      '** read from a database into private members
End Sub
Public Sub IssuePaycheck()
   ...      '** issue employee's paycheck
End Sub
```

Clients have access only to the public members—in this case, `Name`, `Salary`, `ReadFromDB`, and `IssuePaycheck`. These members constitute what is called the default interface of `CEmployee`. In general, an interface is simply *a set of signatures denoting the public properties and methods.* Because a class must expose at least one public member to be useful, this implies that every class in VB has at least one interface—its default.

The key point is that once an interface is published and in use by one or more clients, you should never change it. Doing so will break compatibility with your client base. For example, suppose our `CEmployee` class is compiled in a stand-alone COM component. Now consider the following client code written against `CEmployee`'s default interface:

```
Dim rEmp As CEmployee      '** reference to default interface
Set rEmp = New CEmployee

rEmp.ReadFromDB
txtName.Text = rEmp.Name
txtSalary.Text = Format(rEmp.Salary, "currency")
```

If you were to change the name of `CEmployee`'s public methods or properties and rebuild the COM component, this client code would no longer compile. If the client code was already compiled into an EXE, changing the type of



`Name` or `Salary` and rebuilding the COM component would cause a run-time failure when executing the client. In fact, any change to a public signature represents a change to an interface, and leads, ultimately, to some kind of error in code using that interface.

As a class designer, what changes can you safely make to your components over time? Because clients do not have access to private members, these can be changed at will. Of course, implementation details can also be modified, as long as the result is semantically equivalent. Lastly, note that although you cannot delete public members from an interface, you can *add* properties and methods without breaking compatibility (see rule 2-5 for a complete discussion of compatibility).

Thinking in terms of interfaces, and thus separating interface from implementation, helps you focus on a critical aspect of software development: maintaining compatibility as a system evolves. The next rule encourages you to take this one step further and actually design your classes in terms of explicit, custom interfaces. The result is that your systems become more open to change.

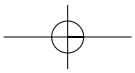
Rule 2-2: Use Custom Interfaces

COM is more than just a technology for building software—it is also a philosophy for building systems that evolve more easily over time. The designers of COM recognized that excessive *coupling* hinders evolution, and thus sought a mechanism that minimized the coupling between components. For example, VB clients typically reference a class directly:

```
Dim rEmp As CEmployee   '** class-based reference = default interface
```

As we know from rule 2-1, this declaration *implicitly* couples the client code to the default interface of class `CEmployee`. Because an interface represents a binding contract between client and class, this coupling prevents the class from evolving. But what if you really need to make an interface change (e.g., to extend a class's functionality or to repair a design oversight)?

The COM solution is to embrace *explicitly* interfaces in both the clients and the classes—an approach known as *interface-based programming.* Instead of presenting a single default interface, classes now publicize one or more *custom* interfaces. Clients then decide which custom interface they need, and couple to



this interface much like before. The key difference, however, is that classes are free to introduce new custom interfaces over time. This allows the class to evolve and to serve new clients, yet remain backward compatible with existing clients. Interface-based programming is thus a design technique in which interfaces serve as the layer of abstraction between clients and classes. As shown in Figure 2.2, this minimizes coupling on the class itself.

How do you define a custom interface? Like classes, custom interfaces are created in VB using class modules. Unlike classes, they contain no implementation because a custom interface is simply a set of method signatures. For example, here's the default interface of `CEmployee` (from rule 2-1) rewritten as a custom interface named `IEmployee`:

```
'** class module IEmployee
Option Explicit

Public Property Get Name() As String
End Sub

Public Property Get Salary() As Currency
End Sub

Public Sub ReadFromDB()
End Sub

Public Sub IssuePaycheck()
End Sub
```

Note the absence of implementation details (i.e., private members and code). A custom interface thus represents an *abstract* class, which is conveyed in VB by setting the class's `Instancing` property to `PublicNotCreatable`. This also prevents clients from mistakenly trying to instantiate your interfaces at runtime.

Figure 2.2 Two views of an interface

Once defined, custom interfaces must be *implemented* in one or more class modules. For example, here is the class `CConsultant` that implements our custom interface `IEmployee`:

```
'** class module CConsultant
Option Explicit

Implements IEmployee

Private sName As String
Private cSalary As Currency

Private Property Get IEmployee_Name() As String
   IEmployee_Name = sName
End Sub

Private Property Get IEmployee_Salary() As Currency
   IEmployee_Salary = cSalary
End Sub

Private Sub IEmployee_ReadFromDB()
   ...      '** read from a database into private members
End Sub

Private Sub IEmployee_IssuePaycheck()
   ...      '** issue employee's paycheck
End Sub
```

Observe that every member in the class is labeled private! Clients thus cannot couple to `CConsultant` in any way, allowing it to evolve freely. Compatibility is maintained by continuing to implement `IEmployee`.

In general, clients now have a choice when accessing a class: to use its default interface or to use any one of the custom interfaces it implements. This choice is expressed by declaring your reference variables of the appropriate interface. For example, here we are accessing a `CConsultant` object through the `IEmployee` interface:

```
Dim rEmp As IEmployee          '** reference to custom interface
Set rEmp = New CConsultant     '** class that implements this interface
```

```
rEmp.ReadFromDB
txtName.Text = rEmp.Name
txtSalary.Text = Format(rEmp.Salary, "currency")
```

This situation is depicted in Figure 2.3. Note that the `CConsultant` object publicizes two interfaces: a default and `IEmployee`. VB classes always define a default interface, enabling clients to use class-based references:

```
Dim rEmp2 As CConsultant   '** class-based reference = default interface
Set rEmp2 = ...
```

This is true regardless of whether the interface is empty, which it is in the case of `CConsultant` because the class contains no public members. The variable `rEmp2` is thus useless, because there are no properties or methods to access.

Now that we can define, implement, and use custom interfaces, you may be wondering: How exactly does all this help me evolve my system more easily? Whenever you need to change a private implementation detail, merely recompile and redeploy the component (be sure to read rule 2-5 before recompiling COM components in VB). And when you need to make an interface change, simply introduce a new custom interface. In other words, suppose you want to evolve the `CConsultant` class by applying some bug fixes as well as by making a few interface changes. You would define a new interface, `IEmployee2`, implement it within `CConsultant`, apply the other bug fixes, recompile, and redeploy.

When existing clients come in contact with instances of the revised class, the result is shown in Figure 2.4 (notice the third lollipop).

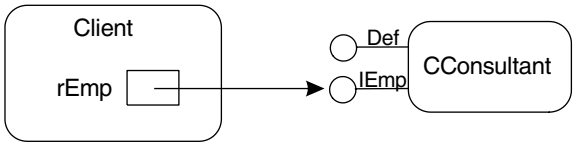


Figure 2.3 Referencing an object through a custom interface

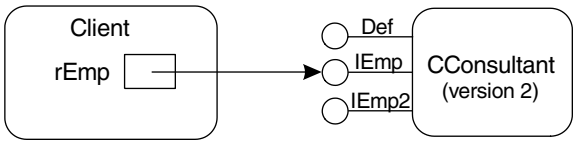


Figure 2.4 An existing client referencing a new version of class `CConsultant`

By introducing new interfaces, classes evolve to support new clients while remaining compatible with existing ones. Note that you have two choices when defining a new interface: It is completely self-contained or it works in conjunction with other interfaces. For example, suppose the motivation for `IEmployee2` is to add parameters to the method `ReadFromDB`, and also to add a method for issuing a bonus. In the first approach, you redefine the entire interface:

```
'** class module IEmployee2 (self-contained)
Option Explicit

Public Property Get Name() As String          '** unchanged
End Sub

Public Property Get Salary() As Currency      '** unchanged
End Sub

Public Sub ReadFromDB(rsCurRecord As ADODB.Recordset)
End Sub

Public Sub IssuePaycheck()                    '** unchanged
End Sub

Public Sub IssueBonus(cAmount As Currency)
End Sub
```

And then your classes implement both. For example, here's the start of the revised `CConsultant` class:

```
'** class module CConsultant (version 2)
Option Explicit

Implements IEmployee
Implements IEmployee2
 .
 .
 .
```

Although the class contains some redundant entry points (`Name`, `Salary`, and `IssuePaycheck` are identical in both interfaces), the advantage is that clients need to reference only one interface—either `IEmployee` or `IEmployee2`. The alternative approach is to *factor* your interfaces, such that each new interface includes only the changes and the additions. In this case, `IEmployee2` would contain just two method signatures:

```
'** class module IEmployee2 (factored)
Option Explicit

Public Sub ReadFromDB(rsCurRecord As ADODB.Recordset)
End Sub

Public Sub IssueBonus(cAmount As Currency)
End Sub
```

This eliminates redundancy in the class, but requires more sophisticated programming in the client. For example, here's the revised client code for reading an employee from a database and displaying their name and salary:

```
Dim rEmp  As IEmployee          '** one reference var per interface
Dim rEmp2 As IEmployee2

Set rEmp  = New CConsultant     '** create object, access using IEmp
Set rEmp2 = rEmp                '** access same object using IEmp2

rEmp2.ReadFromDB ...            '** read from DB/RS using IEmp2
txtName.Text = rEmp.Name        '** access properties using IEmp
txtSalary.Text = Format(rEmp.Salary, "currency")
```

This is depicted in Figure 2.5. Note that both variables reference the same object, albeit through different interfaces.

As your system evolves, different versions of clients and classes may come in contact with one another. For example, it's very common for classes to gain functionality over time, and thus for a single client to interact with numerous iterations of a class. This implies the need for a mechanism by which a compiled client, already deployed in production, can determine what functionality an object provides; i.e., what interfaces it currently implements. Such a mechanism, based on run-time type information (RTTI), is provided by every COM object and is accessed using VB's `TypeOf` function.

Suppose our system contains a number of different employee classes: `CConsultant`, `CTechnical`, `CAdministrative`, and so forth. All such classes implement `IEmployee`, but currently only a few have been revised to implement `IEmployee2`. Now, suppose the task at hand is to send out a bonus to every employee who is not a consultant. Assuming the employee objects are stored in a collection, we can iterate through the collection and simply check the interfaces published by each object:

```
Public Sub SendOutBonuses(colEmployees As Collection, _
                          cAmount As Currency)
   Dim rEmp As IEmployee, rEmp2 As IEmployee2

   For Each rEmp in colEmployees
      If TypeOf rEmp Is CConsultant Then '** no bonus for you
         '** skip
      Else '** issue this employee a bonus...
         If TypeOf rEmp Is IEmployee2 Then '** use interface
            Set rEmp2 = rEmp
            rEmp2.IssueBonus cAmount
```

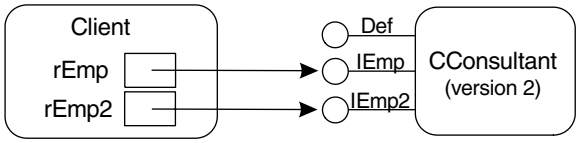


Figure 2.5 Each interface requires its own reference variable in the client

```
            Else '** issue bonus the old-fashioned way
               <human intervention is required>
            End If
         End If
      Next rEmp
   End Sub
```

Even though the default interface `CConsultant` is empty, we use it as a *marker interface* to identify consultants uniquely. Of the remaining employees (all of whom receive a bonus), we check for the `IEmployee2` interface and apply the `IssusBonus` method if appropriate. Failing that, human intervention is required because the employee object does not provide an automatic mechanism. The beauty of `TypeOf` is that it is a run-time mechanism: The next time you execute it, it will respond `True` if the class has been revised to implement that interface. Thus, as more and more classes implement `IEmployee2` over time, `SendOutBonuses` will demand less and less human intervention.

The previous discussion reveals another advantage of custom interfaces—*polymorphism* (see rule 1-7 for a more precise definition). If you think of custom interfaces as reusable designs, then it makes perfect sense for different classes to implement the same interface. This leads to plug-compatible components, and a powerful, polymorphic style of programming in the client in which code is (1) reusable across different classes and (2) resilient to change as classes come and go. For example, consider once again a system with numerous employee classes that all implement `IEmployee`. As implied by Figure 2.6, our client-side code is compatible with any of these employee classes. Thus, if we need to pay everyone, this is easily done using the `IssuePaycheck` method implemented by each class:

```
Public Sub PayEveryone(colEmployees As Collection)
   Dim rEmp As IEmployee

   For Each rEmp in colEmployees
      rEmp.IssuePaycheck
   Next rEmp
End Sub
```

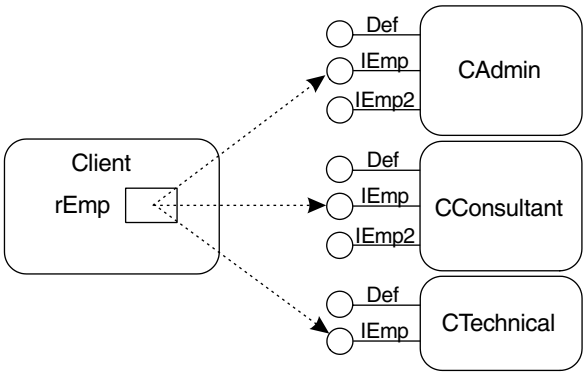


Figure 2.6 Custom interfaces encourage polymorphism

In other words, `IssuePaycheck` is polymorphic and can be applied without concern for the underlying object type. Furthermore, if new employee classes are added to the system, as long as each class implements `IEmployee`, then the previous code will continue to function correctly *without recompilation or modification.* As you can imagine, given a large system with many employee types and varying payment policies, polymorphism becomes a very attractive design technique.

Lest we all run out and start redesigning our systems, note that custom interfaces come at a price. They do require more effort, because each interface is an additional entity that must be maintained. Custom interfaces also complicate the compatibility issue, in the sense that default interfaces are easily extended (as a result of built-in support from VB) whereas custom interfaces are immutable (see rule 2-5 for a detailed discussion of maintaining compatibility in VB). Finally, scripting clients such as Internet Explorer (IE), Active Server Pages (ASP), and Windows Scripting Host (WSH) cannot access custom interfaces directly. They are currently limited to a class's default interface. This last issue is problematic given the importance of scripting clients in relation to the Web. Thankfully, a number of workarounds exist (see rule 4-5) until compiled environments (such as ASP.NET) become available.

Generally, however, the benefits of custom interfaces far outweigh the costs. Custom interfaces force you to separate design from implementation, encour-

aging you to think more carefully about your designs. They facilitate design reuse as well as polymorphism. Of course, custom interfaces also serve to minimize coupling between clients and classes, allowing your classes to evolve more freely while maintaining compatibility. As a result, you'll be able to "field-replace" components as business rules change or bug fixes are applied, insert new components of like behavior without having to revisit client code, and define new behavior without disturbing existing clients. You should thus consider the use of custom interfaces in all your object-oriented systems, but especially large-scale ones in which design and coupling have a dramatic effect.

Custom interfaces are so important that COM is based entirely on interfaces. Clients cannot access COM objects any other way. Hence, COM programmers are interface-based programmers. In fact, there exists a language, the Interface Description Language (IDL), solely for describing interfaces. Often called "the true language of COM," IDL is what allows a COM object developed in programming environment X to be accessed from a client written in programming environment Y. Although typically hidden from VB programmers, there are definite advantages to using IDL explicitly to describe your custom interfaces. Read on; we discuss this further in the next rule.

Rule 2-3: Define Custom Interfaces Separately, Preferably Using IDL

Interface-based programming is a powerful mechanism for building systems that evolve more easily over time. The importance of this programming style is evident in the design of Java, which raised interfaces to the same level as classes. Of course, COM is another compelling example of the significance of interfaces.

A custom interface is nothing more than a set of method signatures defined as a stand-alone entity. As discussed in the previous rule, however, this simple concept enables a wide range of advantages—in design, reuse, and evolution. The proposal here is to take this one step further and to define the interfaces separately.

In VB interfaces are typically defined as `PublicNotCreatable` class modules and are then implemented in `MultiUse` class modules within the same project (see the previous rule if you have never worked with custom

interfaces). Although this is a perfectly adequate approach to implementation, there is a significant drawback: When you hand out your interfaces as class modules, you are essentially giving out the source code. This opens up the possibility that others may change your interfaces, thus altering your design and breaking compatibility with your clients. Whether the changes are accidental or intentional, this is a dangerous loophole.

As shown in Figure 2.7, the simplest precaution is to define your interfaces in a separate ActiveX DLL project, compile and hand out the resulting DLL file. Class implementers then create a *separate* VB project, set a project reference to this DLL, and implement the custom interfaces as before. Likewise, the client (typically denoted by a standard EXE project) must also set a reference to the interface's DLL. This scenario is depicted in Figure 2.8. Note that the client project actually references both the interface's DLL and the class's DLL. The former is needed to declare variables referencing a custom interface, whereas the latter is necessary to instantiate classes using `New`. For example,

```
Dim rEmp As IEmployee          '** need interface information
Set rEmp = New CConsultant     '** new class information
```

Keep in mind that a COM-based DLL must be *registered* on your machine before it can be referenced from a VB project. This can be done using the Windows utility `RegSvr32`, or through the Browse button available off the Project >> References menu item in VB.

Although separating your interfaces into a separate VB project is an improvement, it is somewhat confusing to use an executable DLL to represent

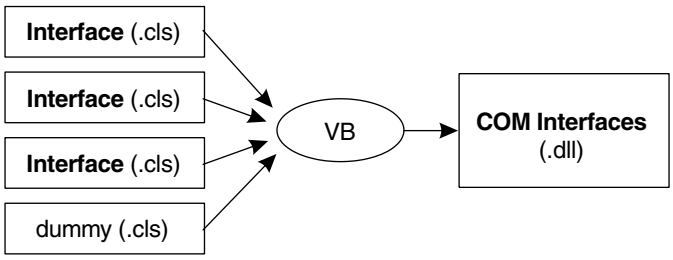


Figure 2.7 Defining custom interfaces separately in VB

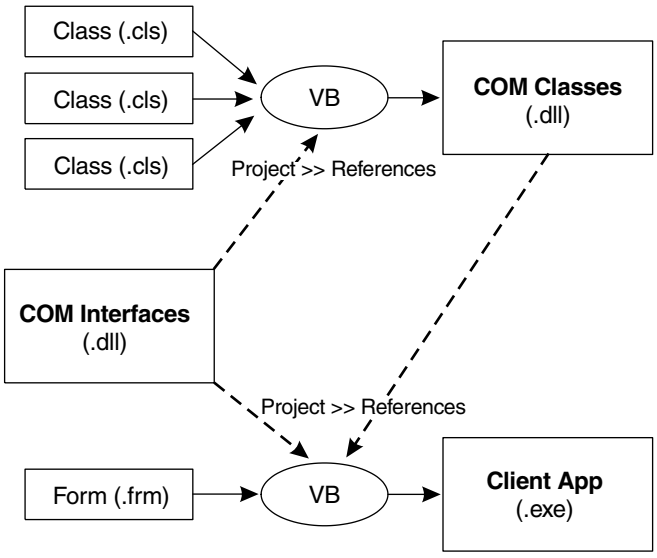


Figure 2.8 Accessing the custom interface's DLL in other VB projects

entities that contain no implementation! Furthermore, because ActiveX DLL projects must contain at least one public class, to make this work you must define a dummy `MultiUse` class along with your `PublicNotCreatable` interface classes (see Figure 2.7).[1] However, the most significant disadvantage to this approach is that VB allows you to extend your custom interfaces, without warning, even though the result breaks compatibility with your clients. For example, accidentally adding a method to an interface and recompiling the interface's DLL will break both the class implementer and the client application. This is true regardless of VB's *compatibility mode* setting when working with your interfaces (see rule 2-5).

The alternative, and better approach, is to do what C++ and Java programmers have been doing for years: defining their interfaces separately using IDL. IDL is a small C-like language solely for describing interfaces, and is often called "the true language of COM" because it is what enables clients and COM

[1] This is not true of ActiveX EXE projects, so you can avoid dummy classes if you want.

components to understand each other. Thus, the idea is to abandon VB, define your interfaces as a text file using IDL, compile this file using Microsoft's IDL compiler (MIDL), and deploy the resulting binary form, known as a *type library* (TLB).[2] This is outlined in Figure 2.9. Once you have a TLB definition of your interfaces, your clients simply set a project reference to the TLB file instead of the interface's DLL file (Figure 2.10). Note that TLBs are registered using the *RegTLib* utility, or via the Browse button under VB's Project >> References.[3]

The advantage to using IDL and MIDL is that you, and only you, can change an interface or break compatibility. You have complete control over your design, and exactly when and how it evolves. The drawback is that you have to learn yet another language. The good news is that we'll show you a way to generate automatically 98 percent of the IDL you'll need. But first, let's take a peek at what IDL looks like. As an example, consider the following VB interface `IEmployee`:

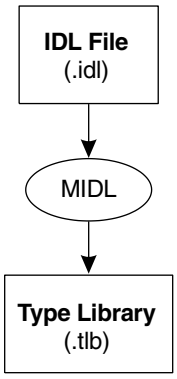


Figure 2.9 Defining custom interfaces separately using IDL

[2] MIDL is a command-line utility that ships as part of Visual Studio (newer versions are available in the Win32 SDK). Another tool, MkTypLib, performs the same function but accepts a slightly different interface language. We recommend the use of MIDL.

[3] Unless otherwise specified, all utilities mentioned here ship with Visual Studio. To provide easy access in a command window, run VCVars32.BAT to set up you path.

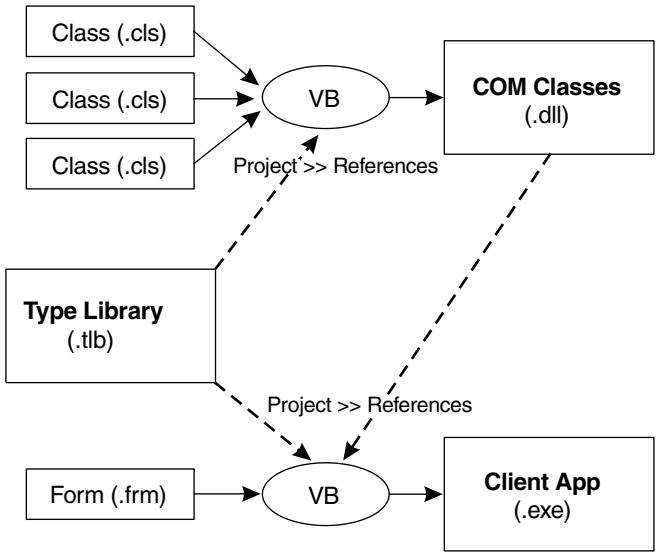


Figure 2.10 Accessing the custom interface's TLB in other VB projects

```
Public Name As String

Public Sub ReadFromDB(rsCurRecord As ADODB.Recordset)
End Sub

Public Function IssuePaycheck() As Currency
End Function
```

To make things more clear, let's first rewrite this as a custom interface (no data members), with parameter passing explicitly defined:

```
Private Property Get Name() As String
End Property

Private Property Let Name(ByVal sRHS As String)
End Property

Public Sub ReadFromDB(ByRef rsCurRec As ADODB.Recordset)
End Sub
```

```
Public Function IssuePaycheck() As Currency
End Function
```

Now, here is the equivalent COM-based interface in IDL:

```
[
   uuid(E1689529-01FD-42EA-9C7D-96A137290BD8),
   version(1.0),
   helpstring("Interfaces type library (v1.0)")
]
library Interfaces
{
   importlib("stdole2.tlb");

   [
      object,
      uuid(E9F57454-9725-4C98-99D3-5F9324A73173),
      oleautomation
   ]
   interface IEmployee : IUnknown {
      [propget] HRESULT Name([out, retval] BSTR* ps);
      [propput] HRESULT Name([in] BSTR s);

      HRESULT ReadFromDB([in, out] _Recordset** pprs);
      HRESULT IssuePaycheck([out, retval] CURRENCY* pc);
   };
};
```

This IDL description defines a TLB named `Interfaces`, which is tagged with three *attributes* (values within square brackets). When registered, the `helpstring` attribute makes the TLB visible to VB programmers as "Interfaces type library (v1.0)," whereas internally it is represented by the *globally unique* identifier (GUID) E1689529-01FD-42EA-9C7D-96A137290BD8 because of the `uuid` attribute. The library contains one interface, `IEmployee`, uniquely identified by the GUID E9F57454-9725=4C98-99D3-5F9324A73173. The remaining attributes define the version of IDL we are using (`object`) and enable automatic proxy/stub generation (`oleautomation`). `IEmployee` consists of four method signatures, each of which is defined as a function returning a 32-bit COM error code (`HRESULT`). Note that VB functions (such as

IssuePaycheck) are redefined to return their values invisibly via an additional out parameter. Finally, for each method, the VB parameter type is translated to the equivalent IDL data type, and the parameter-passing mechanism (ByVal versus ByRef) is transformed to its semantic equivalent (in versus in/out). The most common mappings from VB to IDL data types are shown in Table 2.1.

In general, a TLB may contain any number of interface definitions. When writing IDL, the first step is to assign the TLB and each interface a GUID. GUIDs can be generated using the GuidGen utility: Select Registry Format, press New GUID, then Copy, and paste the resulting GUID into your IDL file. Next, assign the TLB and interfaces the same set of attributes shown earlier. Finally, define each interface. Once you have the IDL file, simply run MIDL to compile it (see Figure 2.9). For example, here's the compilation of Interfaces.idl:

```
midl Interfaces.idl
```

Table 2.1 VB-to-IDL data type mappings

VB	IDL
Byte	unsigned char
Integer	short
Long	long
Single	float
Double	double
Array	SAFEARRAY(<type>) *
Boolean	VARIANT_BOOL
Currency	CURRENCY
Date	DATE
Object	IDispatch *
String	BSTR
Variant	VARIANT

This produces the TLB `Interfaces.tlb`. The interfaces are now ready for use by your clients (see Figure 2.10).[4] Note that your clients do not have to be written in VB. For example, class implementers can use C++ if they prefer. In fact, another advantage of using IDL is that MIDL can automatically generate the additional support files needed by other languages.

Although writing IDL is not hard, it is yet another language that you must learn. Furthermore, you must be careful to use only those IDL constructs and types that are compatible with VB. This is because of the fact that IDL is able to describe interfaces for many different object-oriented programming languages (C++, Java, and so on), but only a subset of these interfaces are usable in VB. Thus, writing IDL from scratch is not a very appealing process for VB programmers.

Luckily, Figure 2.11 presents an easy way to generate VB-compatible IDL automatically. Given a VB ActiveX DLL (or EXE), the `OLEView` utility can be used as a decompiler to reverse engineer the IDL from the DLL's embedded TLB (put there by VB). Obviously, if we start with a DLL built by VB, the resulting IDL should be VB compatible! The first step is to define your interfaces using VB, as discussed earlier (see Figure 2.7). Then, run `OLEView` (one of the Visual Studio tools available via the Start menu) and open your DLL file via File >> View TypeLib. You'll be presented with the reverse-engineered IDL. Save this as an IDL file. Edit the file, defining a new GUID for the TLB as well as for each interface, Enum, and UDT. Now compile the IDL with MIDL, unregister the VB DLL, and register the TLB. In a nutshell, that's it.

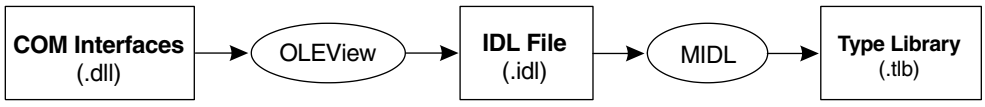


Figure 2.11 Reverse-engineering IDL using `OLEView`

[4] Note that class implementers should continue to work in *binary compatibility* mode. See rule 2-5.

Unfortunately, OLEView is not perfect: You will want to modify the resulting IDL file before compiling it with MIDL. If your interfaces use the VB data type Single, the IDL will incorrectly use Single as well. Search the file and change all occurrences of Single to float. Also, if your interfaces define a UDT called X, the equivalent IDL definition will be incorrect. Manually change struct tagX {...} to struct X {...}. Finally, you'll want to delete some unnecessary text from the IDL file. In particular, it will contain one or more *coclass* (or COM class) definitions, including the dummy class you may have defined for VB to compile your interfaces classes. For example, suppose Interfaces.DLL contains the IEmployee interface class we discussed earlier, as well as a CDummy class. Decompiling the DLL with OLEView yields

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: Interfaces.DLL
[
   uuid(E1689529-01FD-42EA-9C7D-96A137290BD8),
   version(1.0),
   helpstring("Interfaces type library (v1.0)")
]
library Interfaces
{
   // TLib :  // TLib : Microsoft ADO : {...}
   importlib("msado15.DLL");
   // TLib :  // TLib : OLE Automation : {...}
   importlib("stdole2.tlb");

   // Forward declare all types defined in this typelib
   interface _IEmployee;
   interface _CDummy;

   [
      odl,
      uuid(E9F57454-9725-4C98-99D3-5F9324A73173),
      version(1.0),
      hidden,
      dual,
      nonextensible,
      oleautomation
   ]
```

```
    interface _IEmployee : IDispatch {
        [id(0x40030000), propget]
        HRESULT Name([out, retval] BSTR* Name);
        [id(0x40030000), propput]
        HRESULT Name([in] BSTR Name);
        [id(0x60030000)]
        HRESULT ReadFromDB([in, out] _Recordset** );
        [id(0x60030001)]
        HRESULT IssuePaycheck([out, retval] CURRENCY* );
    };

    [ ... ]
    coclass IEmployee {
        [default] interface _IEmployee;
    };

    [ ... ]
    interface _CDummy : IDispatch {
        [id(0x60030000)]
        HRESULT foo();
    };

    [ ... ]
    coclass CDummy {
        [default] interface _CDummy;
    };
};
```

At the very least, your IDL file must contain the code shown in boldface and italic. The rest can be safely deleted. Note that VB defines the name of an interface by starting with the _ character (e.g., `_IEmployee`). Delete this character as well. Next, you should change each interface `odl` attribute to `object`, and `IDispatch` reference to `IUnknown`. Finally, consider adding `helpstring` attributes not only to your interfaces, Enums, and UDTs, but to their individual elements as well. This information is visible when others browse your TLB, yielding a convenient form of documentation.

You are now ready to begin using IDL for defining your interfaces, and to reap the advantages that C++ and Java programmers have been enjoying for years: separating design from implementation, and retaining complete control

over when and how your interfaces change. No one else can alter your design, and only you can break compatibility with clients. The many advantages of custom interfaces rely on your ability to maintain compatibility, and IDL is the best way to go about doing this.

Rule 2-4: Avoid the Limitations of Class-Based Events with Custom Callbacks

The notion of *events* and *event handling* has been a central feature of VB since its inception. The most common case is graphical user interface (GUI) building, which typically requires the programming of form `Load` and `Unload` events, command button `Click` events, and text box `Validate` events. But you can also define and raise your own custom, *class-based* events. For example, the class `CConsultant` could raise a `Changed` event whenever one or more data fields in the object are updated:

```
'** class module CConsultant
Option Explicit

Implements IEmployee

Private sName As String

Public Event Changed()   '** event definition:  Changed

Private Property Get IEmployee_Name() As String
   IEmployee_Name = sName
End Sub

Private Property Let IEmployee_Name(ByVal sRHS As String)
   sName = sRHS
   RaiseEvent Changed    '** name was changed, so raise event!
End Sub
 .
 .
 .
```

Any client that holds a reference to a `CConsultant` object now has the option to handle this event, and thus be notified whenever *that* employee's data changes:

```
'** form module denoting client
Option Explicit

Private WithEvents employee As CConsultant   '** client reference

Private Sub employee_Changed()               '** event handler
   <update form to reflect change in employee object>
End Sub
```

In essence, events enable an object to *call back* to its clients, as shown in Figure 2.12.

Unfortunately, VB's class-based event mechanism has a number of limitations. For the client, the `WithEvents` key word can be applied only to module-level reference variables; arrays, collections, and local variables are not compatible with events. For the class designer, events must be defined in the class that raises them, preventing you from defining a single set of events for reuse across multiple classes. In particular, this means you cannot incorporate events in your custom interfaces, such as `IEmployee`:

```
'** class module IEmployee
Option Explicit

Public Name As String

Public Event Changed()   '** unfortunately, this doesn't work...

Public Sub ReadFromDB(rsCurRecord As ADODB.Recordset)
End Sub

Public Function IssuePaycheck() As Currency
End Function
```

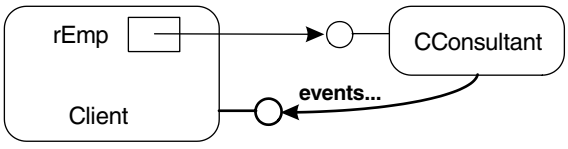


Figure 2.12 Events are really just a callback mechanism

Although VB accepts this event definition, you'll be unable to raise this event from any of your classes.

The solution to these limitations is to design your own event mechanism based on custom interfaces. Consider once again Figure 2.12. Notice that events represent nothing more than an interface implemented by one or more clients. For example, here is a custom interface `IEmployeeEvents` that defines the `Changed` event:

```
'** class module IEmployeeEvents
Option Explicit

Public Sub Changed(rEmp As IEmployee)
End Sub
```

The difference is that events are represented as ordinary subroutines; in this case, with an explicit parameter providing a reference back to the object for ease of access. To receive events, the client now implements the appropriate interface, such as `IEmployeeEvents`:

```
'** form module denoting client
Option Explicit

Implements IEmployeeEvents

Private Sub IEmployeeEvents_Changed(rEmp As IEmployee)
   <update form to reflect change in rEmp.Name, etc.>
End Sub
```

However, we must mention one little detail: How does the object get that reference back to the client, as shown in Figure 2.12? The *object* will raise the event by making a call like this:[5]

```
rClient.Changed Me
```

But who sets this `rClient` reference variable?

[5] If the object executing this code is configured for MTS, you must pass SafeRef(Me).

The client does, by calling the object to set up the callback mechanism. Thus, before any events can occur, the *client* must first call the object and *register* itself:[6]

```
rObject.Register Me     '** register a reference back to ME, the client
```

This means the object must expose a `Register` method. Likewise, the object should expose an `Unregister` method so that clients can stop receiving events:

```
rObject.Unregister Me     '** unregister ME, the client
```

Because every object that wishes to raise events must provide a means of client registration, the best approach is to define these methods in a custom interface and to reuse this design across all your event-raising classes. The following interface `IRegisterClient` summarizes this approach:

```
'** class module IRegisterClient
Option Explicit

Public Enum IRegisterClientErrors
   eIntfNotImplemented = vbObjectError + 8193
   eAlreadyRegistered
   eNotRegistered
End Enum

Public Sub Register(rClient As Object)
End Sub

Public Sub Unregister(rClient As Object)
End Sub
```

Now, every class that wishes to raise events simply implements `IRegisterClient`.

As shown in Figure 2.13, the end result is a pair of custom interfaces, `IRegisterClient` and `IEmployeeEvents`. The object implements

[6] Likewise, if the client executing this code is configured for MTS, pass SafeRef(Me).

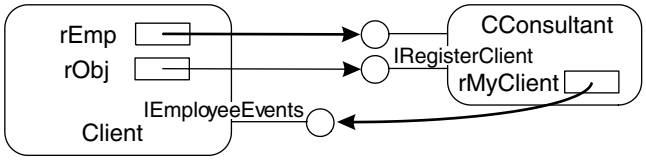


Figure 2.13 An event mechanism based on custom interfaces

`IRegister Client` so that a client can register for events, whereas the client implements `IEmployeeEvents` so the object can call back when the events occur. For completeness, here's the `CConsultant` class revised to take advantage of our custom event mechanism:

```
'** class module CConsultant
Option Explicit

Implements IRegisterClient
Implements IEmployee

Private sName As String
Private rMyClient As IEmployeeEvents   '** ref back to client

Private Sub IRegisterClient_Register(rClient As Object)
   If Not TypeOf rClient Is IEmployeeEvents Then
      Err.Raise eIntfNotImplemented, ...
   ElseIf Not rMyClient Is Nothing Then
      Err.Raise eAlreadyRegistered, ...
   Else
      Set rMyClient = rClient
   End If
End Sub

Private Sub IRegisterClient_Unregister(rClient As Object)
   If Not rMyClient Is rClient Then
      Err.Raise eNotRegistered
   Else
      Set rMyClient = Nothing
   End If
End Sub
```

```
Private Property Get IEmployee_Name() As String
   IEmployee_Name = sName
End Sub

Private Property Let IEmployee_Name(ByVal sRHS As String)
   sName = sRHS

   On Error Resume Next   '** ignore unreachable/problematic clients
   rMyClient.Changed Me   '** name was changed, so raise event!
End Sub
 .
 .
 .
```

The first step is to save the client's reference in a private variable (`rMyClient`) when he registers. Then, whenever we need to raise an event, we simply call the client via this reference. Finally, when the client unregisters, we reset the private variable back to `Nothing`. Note that the `Register` and `Unregister` methods perform error checking to make sure that (1) the client is capable of receiving events, (2) the client is not already registered, and (3) the correct client is being unregistered. Furthermore, to handle multiple clients, also note that the previous approach is easily generalized by replacing `rMyClient` with a collection of client references.

To complete the example, let's assume on the client side that we have a VB form object that instantiates a number of employee objects (`CConsultant`, `CTechnical`, `CAdministrative`, and so on) and displays them on the screen. The client's first responsibility is to implement the custom `IEmployee-Events` interface so it can receive the `Changed` event:

```
'** form module denoting client
Option Explicit

Private colEmployees As New Collection '** collection of object refs

Implements IEmployeeEvents

Private Sub IEmployeeEvents_Changed(rEmp As IEmployee)
   <update form to reflect change in rEmp.Name, etc.>
End Sub
```

Here the Changed event is used to drive the updating of the form. Before the client can receive these events however, it must register with each employee object *that supports "eventing."* In this case we assume the employees are created during the form's Load event based on records from a database:

```
Private Sub Form_Load()
   <open DB and retrieve a RS of employee records>

   Dim rEmp As IEmployee, rObj As IRegisterClient

   Do While Not rsEmployees.EOF
      Set rEmp =CreateObject(rsEmployees("ProgID").Value)

      If TypeOf rEmp Is IRegisterClient Then '** event based
         Set rObj = rEmp    '** switch to register interface...
         rObj.Register Me   '** and register myself to receive events
      End If

      rEmp.ReadFromDB rsEmployees
      colEmployees.Add rEmp
      rsEmployees.MoveNext
   Loop

   <close DB and RS>
End Sub
```

Lastly, the client is also responsible for unregistering when it no longer wishes to receive events. This task is performed during form Unload (i.e., when the form is no longer visible):

```
Private Sub Form_Unload(Cancel As Integer)
   Dim rEmp As IEmployee, rObj As IRegisterClient
   Dim l As Long

   For l = colEmployees.Count To 1 Step -1
      Set rEmp = colEmployees.Item(l)
      colEmployees.Remove l
```

```
        If TypeOf rEmp Is IRegisterClient Then '** event based
           Set rObj = rEmp       '** switch to register interface...
           rObj.Unregister Me    '** and unregister myself
        End If
     Next l
End Sub
```

Note that unregistering is important to break the cyclic reference formed between the client and the object.

Although custom callbacks require more effort to set up than VBs built-in event mechanism, the benefits are many: reusable designs, better callback performance, and more flexibility during implementation.[7] For example, an object with multiple clients can apply a priority-based event notification scheme if desired. In the greater scheme of things, custom callbacks also illustrate the power of using interfaces to design flexible solutions to everyday problems.

Rule 2-5: Be Deliberate About Maintaining Compatibility

In a COM-based system, clients communicate with objects via interfaces. These interfaces must be well-defined, registered, and agreed on by all parties for your system to run properly (Figure 2.14). The good news is that this is relatively easy to ensure in your first release: Recompile the COM servers, then recompile the clients and deploy.

However, at some point you will be faced with recompiling and redeploying one of your COM servers—perhaps to apply bug fixes or to add new functionality. In this case, what happens to the clients? You can either (1) redeploy all new clients to match or (2) ensure that your COM server maintains *compatibility* with the existing clients. Although the latter is typically preferred (and certainly less work), it requires that you have a solid understanding of COM's rules for *versioning,* and how VB applies those rules. Otherwise, recompiling a COM server can lead to all sorts of errors on the client side, from "Can't create object" and "Type mismatch" to the dreaded GPF.

[7] Keep in mind that security may be an issue if raising events across processes/machines, because the object needs permission to call the client.

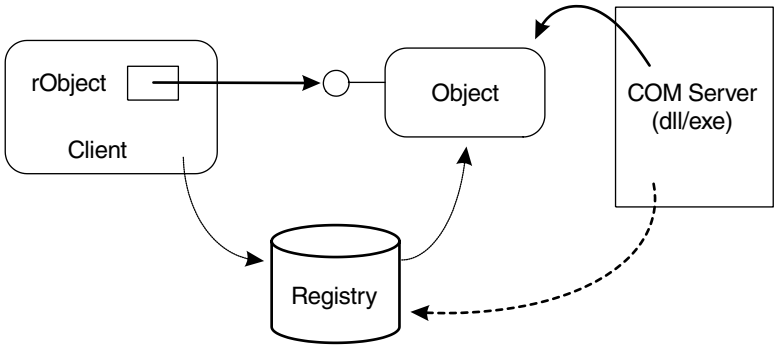


Figure 2.14 COM requires that servers (and their interfaces) be registered

Before we start, let's review some important concepts in COM. A typical COM server is a DLL/EXE that defines one or more classes, one or more interfaces, and a TLB that summarizes this information. Every class, interface, and TLB is assigned a unique 128-bit integer called a *GUID.* These are referred to as *CLSIDs*, *IIDs,* and *LibIDs,* respectively. GUIDs are compiled in the COM server that defines them, make their way into the registry when the COM server is registered, and usually get compiled in the clients as well. COM *activation* is the process of creating an instance of a class from a COM server, triggered, for example, when a client executes `New`. To activate, COM requires both a CLSID and an IID, locates the COM server via the registry, asks the COM server to create the instance, and then obtains the proper interface reference for return to the client. As discussed in rule 2-3, you can use the `OLEView` utility to view the contents of a server's TLB and to see the GUIDs firsthand.

Lastly, it's very important to understand the difference between a default interface and a custom one. Review rules 2-1 and 2-2 if necessary.

To maintain compatibility with clients, the short answer is that when recompiling a COM server, you need to focus on three things: functionality, interfaces, and GUIDs. Obviously, although implementation details may change, the server's overall functionality must be compatible from one version to the next. Second, the interfaces exposed by each class should not change in any way. Methods cannot be deleted, their names cannot differ, and their parameters

cannot vary (not in number, type, or order). Finally, the identifying GUIDs should not change (i.e., the CLSIDs, IIDs, and LibID). Let's look at these compatibility issues in more detail.

Scripting Clients

The first step is to understand your clients. There are two types: *scripting* and *compiled.* Scripting clients are typically written in VBScript or JavaScript and are executed in environments such as ASP, IE, or WSH. The key characteristic of a scripting client is its use of generic object references:

```
Dim rObj      '** As Variant / Object
```

This typeless variable represents a *late-bound* (indirect, less efficient) connection to an object's *default interface.*[8,9] In addition, scripting clients typically create objects using VB's `CreateObject` function, passing the appropriate ProgID (a string denoting the TLB followed by a class name):

```
Set rObj = CreateObject("Employees.CConsultant")
```

`CreateObject` first converts the ProgID to a CLSID (via the registry), and then performs a standard COM activation.[10] Once activated, a scripting client may call any method in the object's default interface. For example,

```
rObj.IssuePaycheck
```

This assumes that `IssuePaycheck` is a public subroutine within class `CConsultant`.

Thus, maintaining compatibility in your COM server amounts to preserving the ProgIDs and the default interfaces. The ProgIDs are easy to deal with: Simply do not change the name of your TLB or your classes. When building COM servers in VB, note that your TLB's name is derived from your VB project's

[8] Every method call typically requires two calls (`lookup` then `invoke`), with variants used to pass parameters. This approach is based on COM's *`IDispatch`* interface.

[9] As discussed in rule 4-2, scripting clients cannot access custom interfaces.

[10] Where does `CreateObject` get the IID? It uses the well-known IID for *`IDispatch`*, COM's predefined late-bound interface. This maps to the default interface of a VB class.

name (a project property). As for the default interfaces, for each class you cannot delete any public subroutine or function, nor can you change its method signature. However, note that because clients are late-bound and parameters are thus passed as variants, it is possible to change a parameter's type in some cases and still maintain compatibility. For example, suppose a class originally contained the following method:

```
Public Sub SomeMethod(ByVal iValue As Integer)
```

This can evolve to

```
Public Sub SomeMethod(ByVal lValue As Long)
```

without breaking compatibility because `Integer` is upward compatible with `Long`.

Finally, it is worth noting that compiled environments also behave like a scripting client when object references are generic. In VB, this occurs whenever clients use the `Variant` or `Object` data type:

```
Dim rObj2 As Object    '** this says I want to be late-bound
Dim rObj3 As Variant   '** likewise...
```

Each reference denotes a late-bound connection to an object, regardless of how that object is created:

```
Set rObj2 = New Employees.CConsultant
Set rObj3 = CreateObject("Employees.CConsultant")
```

In this case, the same compatibility rules apply, with the exception that the client's use of `New` requires that the COM server's CLSIDs and default IIDs also remain unchanged. This is discussed in the next section.

Compiled Clients

Compiled clients are characterized by object references of a specific interface type, for example:

```
Dim rObj4 As Employees.IEmployee      '** a custom interface
Dim rObj5 As Employees.CConsultant    '** the default interface
```

These variables represent a *vtable-bound* (direct, efficient) connection to a specific interface of an object. These interface types must be defined by your COM server—or more precisely, in its TLB—which the client must reference. Object creation is typically done using `New` or `CreateObject`:

```
Set rObj4 = CreateObject("Employees.CConsultant")
Set rObj5 = New Employees.CConsultant
```

Regardless of how the objects are created, at this point the reference `rObj4` can be used to call methods in the custom interface `IEmployee`, whereas `rObj5` can be used to call methods in `CConsultant`'s default interface.

From the perspective of compatibility, the key observation about compiled clients is that they refer to interfaces and classes by name. As a result, when the client code is compiled, the corresponding IIDs and CLSIDs are embedded into the resulting EXE. Thus, maintaining compatibility with compiled clients requires that you preserve not only the ProgIDs and the interfaces, but the GUIDs as well.

Much like scripting clients, the ProgIDs and interfaces are under your control. However, VB is in charge of generating the necessary GUIDs whenever you compile your COM server. So how can you prevent VB from changing these values during recompilation? By manipulating your project's *version compatibility* setting, as shown in Figure 2.15.

The first setting, No Compatibility, means precisely that. If you recompile, all GUIDs will be changed, thereby breaking compatibility with compiled clients. This setting lets you intentionally break compatibility (e.g., when you need to begin a new development effort). The second setting, Project Compatibility, is meant to preserve compatibility with other *developers*. In this case the LibID and CLSIDs are preserved, but the IIDs change. This allows references to your TLB to remain valid (i.e., references to your COM server from other VB projects), as well as class references embedded in Web pages. However, the IIDs continue to change, reflecting the fact that the server is still under construction. The rationale for this setting is team development, and thus the setting should be used when you are developing classes that you must share with others before the design is complete. To help track versions, note that VB changes the version number of your COM server's TLB by a factor of one each

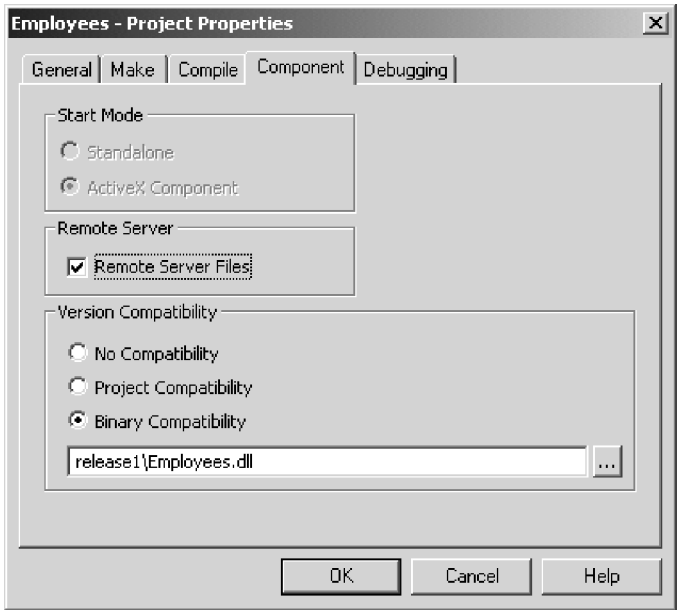


Figure 2.15 VB's version compatibility settings

time you recompile. The third and final setting is Binary Compatibility, in which all GUIDs are preserved from one compilation to the next. This is VB's "deployment" setting, because it enables you to maintain compatibility with compiled clients out in production. Thus, you should switch to binary compatibility mode (and remain there) as soon as you release the first version of your COM server. Note that binary compatibility is necessary even if your interfaces and IIDs are defined separately, because of the fact that your clients may be dependent on the default interfaces generated by VB.[11]

When working in binary compatibility, it's important to understand that VB needs a copy of your released DLL/EXE to maintain compatibility when you recompile. Notice the reference in Figure 2.15 to "release1\Employees.DLL." VB simply copies the GUIDs from the referenced file and uses them to generate the new COM server. It's considered good practice to build each release in

[11] For example, see using IDL as discussed in rule 2-3.

a separate directory (release 1, release 2, and so on) so that you can always recompile against an earlier version if necessary.[12] In general, however, make sure your binary compatibility setting always references the most recent production release. To prevent accidental overwriting, it's also a good idea to keep your release DLLs/EXEs in a version control system for read-only checkout.

Besides retaining GUIDs, binary compatibility mode also protects your interfaces. In particular, VB prevents you from making any changes that might break compatibility. For example, changing a method's parameter type from `Integer`

```
Public Sub SomeMethod(ByVal iValue As Integer)
```

to `Long`

```
Public Sub SomeMethod(ByVal lValue As Long)
```

yields the warning dialog shown in Figure 2.16. At this point, unless you are absolutely sure of what you are doing, you should *cancel* and then either restore the method signature, switch compatibility mode, or define a new interface containing your change.[13] Note that variants can be used as parameter types to give you some flexibility for future evolution without the need to change explicitly the type in the interface.

Version-Compatible Interfaces

The COM purist would argue that when you need to change an interface, you do so by defining a completely new one. This makes versioning easier to track, because each interface will have a distinct name (and IID). Although this may lead to more work within your COM servers, it enables a client to differentiate between versions, and thus remain backward compatible with your earlier COM servers. For example, a client can test for version 2 of the `IEmployee` interface before trying to use it:

[12] For example, if you accidentally break compatibility, you can rereference the proper release version in binary compatibility mode to restore compatibility.

[13] The option to Break compatibility causes all the IIDs to change (much like project compatibility), whereas Preserve compatibility retains the IIDs. Either way, the physical interface is changed. The latter could be selected, for example, if no one is calling `SomeMethod`.

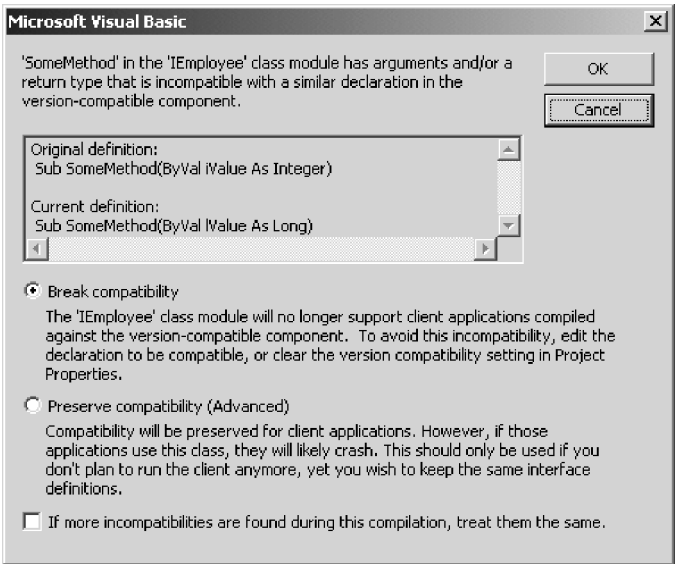


Figure 2.16 VB's warning dialog that an interface has changed

```
If TypeOf rEmp Is IEmployee2 Then '** is v2 available in this object?
   Dim rEmp2 As IEmployee2
   Set rEmp2 = rEmp

   <use rEmp2 to access v2 of IEmployee interface>

   Set rEmp2 = Nothing
End If
```

As a result, new clients can be released before servers are upgraded, or can continue to function properly if servers are downgraded for some reason.

However, although COM purists argue in favor of maintaining *version-identical* interfaces, VB implements a more flexible (but dangerous) notion known as *version-compatible* interfaces. In short, VB's binary compatibility mode actually allows one type of interface change: You may *add* methods to the default interface. When you do so, VB is careful to add the new methods to the end of the class's underlying vtable, generating a single default interface that is compatible with both old and new clients. Note that VB increases the

version number of your COM server's TLB by 0.1 to reflect the fact that the interface changed.

Interestingly, VB isn't breaking the rules of COM, because it generates a new IID to identify the resulting interface. To maintain compatibility, VB must produce code within the COM server so that objects recognize both the new and the old IIDs when queried at run-time. Likewise, the registry must be reconfigured to support the fact that multiple IIDs map to the same physical interface. In particular, the original interface *forwards* to the new interface, as shown in Figure 2.17. Note that interface forwarding is direct. If you add a method from release 1 to release 2, and then add another method in release 3, releases 1 and 2 both forward to release 3.

Why does VB offer this feature? To make it easier for your classes to evolve. Why is this feature dangerous? First of all, there is only one version of an interface from the perspective of the client—the most recent one. VB clients thus cannot use `TypeOf` to determine which version of an interface is available. This makes it harder for clients to achieve backward compatibility with earlier versions of your COM server. Second, VB only provides support for extending the default interface. You cannot add methods to custom interfaces such as `IEmployee`. In fact, adding methods to a custom interface yields no warning from VB, yet breaks compatibility with your clients (even in binary compatibility mode!). Finally, some client-side setup programs fail to register properly the necessary interface forwarding information, leading to COM activation errors at run-time. This is a known problem (e.g., with MTS's `export` command).[14]

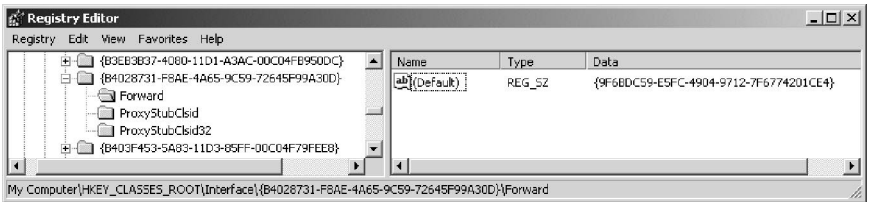


Figure 2.17 Forwarding to a version-compatible interface

[14] See Microsoft Knowledge Base (KB) article Q241637 in the MSDN library.

The safer alternative is that of the COM purist: Use custom interfaces, and define a new interface whenever changes are needed from one release to another. Note that avoiding these dangers is also one of the reasons we recommend defining your custom interfaces outside VB. See rule 2-3 for more details.[15]

COM is a somewhat fragile system, requiring that all participants agree—servers, clients, and registries alike. Because most applications live beyond version 1, maintaining compatibility in the presence of evolution and recompilation becomes one of the most important aspects of COM programming. Although the nuances may be complex, the overall solution is straightforward: Develop in project compatibility, deploy in binary compatibility, and use custom interfaces whenever possible.

Rule 2-6: Choose the Right COM Activation Technique

In VB, the traditional mechanism for creating an object is the `New` operator. For example,

```
Set rEmp = New Employees.CConsultant
```

creates a new instance of the `CConsultant` class. However, this is not the only alternative. If the class is registered as a COM component, you can also use `CreateObject` and `GetObject`:

```
Set rEmp2 = CreateObject("Employees.CConsultant")
Set rEmp3 = GetObject("", "Employees.CConsultant")
```

On the other hand, if the class is configured to run under MTS, then you should probably be using `CreateInstance` instead:

```
Set rEmp4 = GetObjectContext.CreateInstance( _
                              "Employees.CConsultant")
```

[15] See also Microsoft KB articles Q190078, Q190967, and Q191214.

Finally, as if this wasn't enough, when you are writing server-side ASP code you will want to use `Server.CreateObject`:

```
Set rEmp5 = Server.CreateObject( _
                                "Employees.CConsultant")
```

Do you know when to use each technique? If not, then read on . . .

First off, let's clear up a common misconception about how clients bind to objects at run-time.[16] Logically, the situation is depicted in Figure 2.18. Clients hold references to interfaces and use these references to access objects. Physically, however, the binding mechanism used between client and object can vary, *depending on how the reference variables are declared in the client.* There are two main approaches: *vtable-binding* and *late-binding*. The former is more efficient, because the client is bound directly to an interface's implementation. This is available only in compiled environments such as VB, VC++, and soon ASP.Net. For example, the following two declarations dictate vtable-binding (1) to the default interface of class `CConsultant` and (2) to the custom interface `IEmployee`:

```
Dim rEmp1 As Employees.CConsultant   '** (1) default interface
Dim rEmp2 As Employees.IEmployee     '** (2) custom interface
```

This is true regardless of how the objects are created. Regardless of whether the client uses `New` or `CreateObject` or some other mechanism,

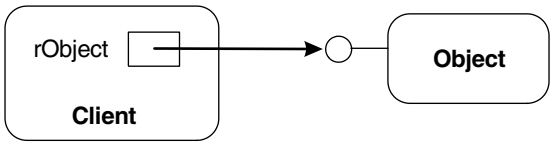
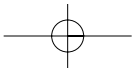


Figure 2.18 A client holding a reference to an object

[16] This discussion is a summary of the more detailed scripting client versus compiled client discussion held earlier in rule 2-5. Rule 2-5 should be read first if COM is new to you.



rEmp1 and rEmp2 are vtable-bound to the objects they reference. In contrast, consider the following declarations:

```
Dim rEmp3 As Variant
Dim rEmp4 As Object
Dim rEmp5                '** implies Variant
```

These references all dictate late-binding, a less efficient mechanism based on COM's IDispatch interface (and one that always maps to the object's default interface). Again, this is true regardless of how the objects are created. For example, any use of rEmp5 is late-bound, even if the object is created using New:

```
Set rEmp5 = New Employees.CConsultant
rEmp5.SomeMethod  '** this implies a lookup of SomeMethod, then invoke
```

Although late-binding is possible in compiled environments like VB (using the previous declarations), it is the only binding technique available in scripting environments such as IE, ASP, and WSH.

Note that the object being created must support the type of binding requested by the client; otherwise a run-time error occurs. Objects built with VB automatically support both vtable-binding and late-binding.

COM Activation

COM activation is the process by which a client creates a COM object at run-time. It is a somewhat complex process that involves the client, GUIDs, the COM infrastructure, one or more registries, and the COM server. Although the details are interesting, what's important here are the goals of activation: (1) create the object and (2) obtain the necessary interface references.[17] Keep in mind that objects can be activated across process and machine boundaries, a daunting task that is automatically handled by the COM infrastructure.

[17] The interested reader is encouraged to read one of the many good books on COm, e.g., *Programming Distributed Applications with COM+ and VB6* by our own Ted Pattison, (MS Press, 2001).

The `New` Operator

The most important characteristic of `New` is that it does not always trigger COM activation. In some cases, a call to `New` results in an optimized form of object creation performed entirely by VB.[18] How the `New` operator behaves depends on whether the class is internal or external, *from the perspective of the client creating the object.* For example, consider the following client code:

```
Dim rObj1 As IInterface
Set rObj1 = New CClass
```

The call to `New` results in VB's optimized creation if the class is internal, (i.e., `CClass` is either (1) part of the same VB project/DLL/EXE as the client, or (2) part of the same VB group as the client [and you are running that group inside the VB IDE]). Otherwise, the class is considered external, and COM activation is performed in an attempt to instantiate the object.

Being aware of `New`'s optimized behavior is important for two reasons. First, it is much more efficient than COM activation, and thus is preferable for performance reasons. But, second, it is incorrect in certain situations, for example, when the class being instantiated is configured to run under MTS or COM+. In this case, COM activation is required for the class to receive the necessary MTS/COM+ services, but if the class is internal then `New` bypasses COM activation, creating an object that may not run properly. For this reason, the conservative programmer should avoid the use of `New`.

Note that the `New` operator can be applied in two different ways: *traditional* and *shortcut.* With the traditional approach, you declare a reference and then create the object separately as needed:

```
Dim rObj2 As IInterface
 .
 .
 .
Set rObj2 = New CClass
rObj2.SomeMethod
```

[18] This is analogous to calling `New` in other object-oriented programming languages like C++, versus calling `CoCreateInstanceEx` in Windows to perform COM activation.

This allows your references to be of any interface type, and makes object creation visible in the code. The second alternative is the shortcut approach, in which you embed the `New` operator in the variable declaration:

```
Dim rObj3 As New CClass
 .
 .
 .
rObj3.SomeMethod
```

In this case, VB automatically creates an object on the first use of the reference variable (`rObj2`). Although this requires less typing, this approach restricts you to a class's default interface, and can lead to interesting runtime behavior. For example, consider the following code fragment:

```
Set rObj3 = Nothing
rObj3.SomeMethod          '** traditionally, this would fail
 .
 .
 .
Set rObj3 = Nothing
If rObj3 Is Nothing Then  '** traditionally, this would be true
   Msgbox "you'll never see this dialog"
End If
```

Each time you use a shortcut reference in a statement, VB first checks to see if the reference is `Nothing`. If so, it creates an object before executing the statement. Not only does this result in additional overhead, but it also prevents you from checking whether an object has been destroyed (the act of testing re-creates another object!). For these reasons, we generally recommend that you avoid the shortcut approach.

CreateObject

Unlike `New`, the `CreateObject` function always creates objects using COM activation. You supply a string-based ProgID, and `CreateObject` converts this

to a CLSID (via a registry lookup) before performing a standard COM activation. Here's a generic example:

```
Dim rObj1 As TLibName.IInterface
Set rObj1 = CreateObject("TLibName.CClass")
```

The advantage to this approach is flexibility. First, because `CreateObject` is based on strings and not class names, the class to be instantiated can be computed at run-time based on user input, configuration files, or records in a database. Second, `CreateObject` has an optional parameter for specifying *where* to create the object (i.e., on which remote server machine). This overrides the local registry settings, once again providing more flexibility at run-time. For example, this feature can be used to implement simple schemes for fault tolerance:

```
On Error Resume Next
Dim rObj2 As TLibName.IInterface
Set rObj2 = CreateObject("TLibName.CClass", "Server1")

If rObj2 Is Nothing Then  '** server1 is down, try server2...
   Set rObj2 = CreateObject("TLibName.CClass", "Server2")
End If

If rObj2 Is Nothing Then  '** both servers are down, give up...
   On Error Goto 0        '** disable local error handling
   Err.Raise ...          '** inform the client
End If

On Error Goto Handler  '** success, reset error handler and begin...
 .
 .
 .
```

Note that the machine names are also string based, and thus can be read from configuration files or a database.

Because `CreateObject` only performs COM activation, it cannot instantiate `Private` or `PublicNotCreatable` VB classes. The class must be a registered COM object. Furthermore, whether you are using vtable-binding or

late-binding, instantiation via `CreateObject` requires that the object support late-binding.[19] If the object does not, VB raises error 429. In these cases, the only way to instantiate the object is via `New`.

`GetObject`

As the name implies, `GetObject` is designed to gain access to existing objects; for example, an MS Word document object in a file:

```
Dim rDoc As Word.Document '**set a  reference to MS Word Object Library
Set rDoc = GetObject("C:\DOCS\file.doc", "Word.Document")
rDoc.Activate
```

However, it can also be used to create new objects. For example,

```
Dim rObj1 As TLibName.IInterface
Set rObj1 = GetObject("", "TLibName.CClass")
```

In this sense, `GetObject` is equivalent to `CreateObject`, albeit without the ability to specify a remote server name.

Interestingly, as we'll see shortly, there's a version of `GetObject` that is more efficient than `CreateObject`, yet it is rarely used for this reason. Instead, it is commonly used to access MS Office objects or Windows services such as Active Directory, Windows Management Instrumentation (WMI), and the Internet Information Server (IIS) metabase. It is also used in conjunction with Windows 2000-based queued components (i.e., objects with method calls that are translated into queued messages for asynchronous processing). For example, suppose the class `CQClass` is configured as a queued component under COM+ on Windows 2000. The following gains access to the appropriate queue object for queuing of method calls (versus creating a traditional COM object that executes the method calls):

```
Dim rQObj As TLibName.CQClass
Set rQObj = GetObject("Queue:/new:TLibName.CQClass")
```

[19] In other words, the object must implement `IDispatch`; some ATL components do not.

```
rQObj.SomeMethod                 '** call to SomeMethod is queued
rQObj.SomeMethod2 "parameter"    '** call to SomeMethod2 is queued

MsgBox "client is done"
```

In this case, the calls to `SomeMethod` and `SomeMethod2` are queued for later processing by some instance of `CQClass`. This implies that a `MsgBox` dialog appears on the screen long before the actual method calls take place.

`GetObjectContext.CreateInstance` and `Server.CreateObject`

Suppose you have two classes configured to run under MTS, `CRoot` and `CHelper`. If `CRoot` needs to create an instance of `CHelper`, then there is exactly one way for `CRoot` to instantiate this class properly—via `GetObject-Context.CreateInstance`:

```
'** code for configured class CRoot
Dim rObj1 As TLibName.IInterface
Set rObj1 = GetObjectContext.CreateInstance( _
                                    "TLibName.CHelper")
```

Likewise, if `CRoot` is an ASP page, then the proper way to instantiate `CHelper` is using `Server.CreateObject`:

```
'** code for ASP page
Dim rObj2
Set rObj2 = Server.CreateObject("TLibName.CHelper")
```

These methods are essentially wrappers around `CreateObject`, accepting a ProgID and performing COM activation. However, they enable the surrounding environment (MTS and ASP respectively) to recognize and to participate in the creation of the COM object. Direct calls to `New` and `CreateObject` bypass the surrounding environment, leading to slower or incorrect execution.[20]

[20] With COM+, it is safe to use `CreateObject` as well as `CreateInstance`. With ASP, use `CreateObject` in cases in which `Server.CreateObject` is designed to fail. See Microsoft knowledge base (KB) article Q193230 in the MSDN library.

For more details on the rationale and proper use of `CreateInstance`, see rule 3-3.

Performance Considerations

Most discussions involving the performance of COM objects focus on two things: (1) the type of binding (vtable versus late) and (2) the marshaling characteristics of any parameters. Although these are very important, little attention is paid to the cost of COM activation. Thus, assuming there is no compelling design reason to choose between `New`, `CreateObject`, and `GetObject`, is there a performance reason?

First, keep in mind that `New` is optimized for internal classes, so it is always the most efficient mechanism when COM activation is not needed. However, let's assume our goal is COM activation. There are three types of activation: *in-process, local,* and *remote.* In-process activation means the resulting object resides in the same process as the client. Both local and remote activation represent *out-of-process* activation, in which the object resides in a process separate from the client—either on the same machine (local) or a different one (remote). Examples of in-process activation include classes packaged as an ActiveX DLL and then registered as COM objects, and classes configured to run under MTS as a library package. Examples of local and remote activation include classes packaged in an ActiveX EXE, and classes configured to run under MTS as a server package.

In the case of in-process activation, `New` is always the most efficient: It is 10 times faster than `CreateObject` and is 10 to 20 times faster than `GetObject`. This is mainly the result of the fact that `CreateObject` and `GetObject` require additional steps (e.g., the conversion of the ProgID to a CLSID). Interestingly, in the out-of-process cases, the best performer varies: `New` is more efficient (10 percent) when you plan to use vtable-binding against the object's default interface, whereas `CreateObject` and `GetObject` are more efficient when you plan to use late-binding against the default interface (two times) or vtable-binding against a custom interface (10 to 15 percent). Let's discuss why this is so.

As noted earlier, COM activation has two goals: (1) create the object and (2) acquire the necessary interface references. The `New` operator is essentially

optimized for vtable-binding against an object's default interface. In one API call, `New` creates the object and acquires four interface references: the default interface, `IUnknown`, `IPersistStreamInit`, and `IPersistPropertyBag`. On the other hand, `CreateObject` and `GetObject` are optimized for late-binding, because they acquire a slightly different set of interface references: `IDispatch`, `IUnknown`, `IPersistStreamInit`, and `IPersistPropertyBag`. Note that `CreateObject` and `GetObject` also take longer to create the object and to acquire these references (two API calls and three method calls).

So why is `New` slower in some cases? Recall that out-of-process activation yields proxy and stub objects to handle the communication between client and object (Figure 2.19). A proxy/stub pair is created during activation for each interface that is acquired, and thus forms part of the activation cost. Assuming the object does not perform custom marshaling,[21] the proxy-stub pair associated with its default interface is much more expensive to create than those associated with predefined COM interfaces such as `IDispatch`. As a result, if the client ends up using the default interface, then `New` is faster because it automatically acquires a reference to the object's default interface. However, if the client needs `IDispatch` (late-binding) or a custom interface, then `CreateObject` and `GetObject` are faster because time is not wasted building an expensive proxy/stub pair that will never be used. The results are summarized in Table 2.2.

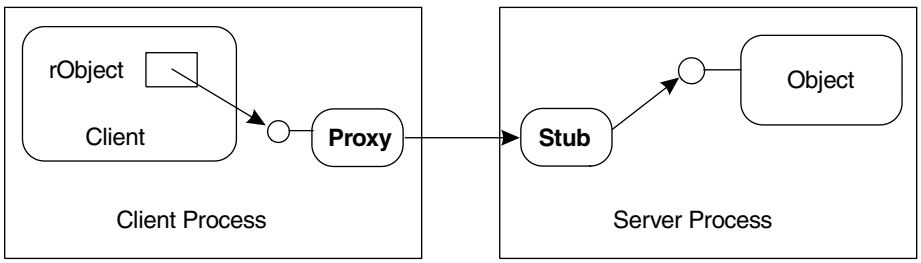


Figure 2.19 COM out-of-process activation yields proxy and stub objects

[21] In other words, the object uses COM's standard marshaling infrastructure. By default, VB objects rely on standard marshaling.

Table 2.2 Maximizing performance of COM objects

Activation Type	Interface Client Will Use	No. of Calls Client Will Make	Best Performance	
			Activation	Binding
In-process	—	—	`New`	vtable
Local	default	< 10	`CreateObject`	late
		³ 10	`New`	vtable
	custom	—	`CreateObject`	vtable
Remote	default	< 3	`CreateObject`	late
		³ 3	`New`	vtable
	custom	—	`CreateObject`	vtable

What's fascinating is that activation is not the complete picture. The conventional wisdom for best overall performance is to access the object using vtable-binding because it requires fewer actual calls to the object and passes parameters more efficiently. However, vtable-binding implies the direct use of an object's interface (default or custom), and hence the need for an expensive proxy/stub pair in the out-of-process case. For example, assume the following client-side code is activating an out-of-process COM object:

```
Dim rObj1 As TLibName.CClass   '** implies default interface
Set rObj1 = CreateObject("TLibName.CClass")
```

Even though `CreateObject` avoids the expensive proxy/stub pair, the `Set` statement will trigger their creation because the type of the variable being assigned is one of the object's interfaces. Therefore, to get the full benefit of using `CreateObject`, it turns out that you must also use late-binding! In other words,

```
Dim rObj2 As Object   '** implies IDispatch to default interface
Set rObj2 = CreateObject("TLibName.CClass")
```

is roughly twice as fast as the previous code fragment. Of course, late-binding is more expensive per call, and thus the advantage of this approach diminishes as the number of calls increases. This explains the results in Table 2.2, in which there exists some threshold at which point vtable-binding becomes more efficient. Note that the exact threshold will vary in different situations (based on network speeds and distances, interface designs, and so on).

Lastly, if you are running Windows 2000 and are truly concerned with performance, you might consider using `GetObject` in place of `CreateObject`. `GetObject` is slightly more efficient when used as follows:

```
Dim rObj As ...
Set rObj = GetObject("new:TLibName.CClass")
```

In this case, `GetObject` acquires only two interface references instead of four; namely, `IDispatch` and `IUnknown`. Although this speeds up activation by reducing the number of method calls (which may be traversing across the network), it prevents the proper activation of "persistable" objects because `IPersistStreamInit` and `IPersistPropertyBag` are no longer available.

Fortunately or unfortunately, VB offers a number of different techniques for creating objects. Some always perform COM activation (`CreateObject` and `GetObject`); some do not (`New`). Some are more flexible (`CreateObject` and `GetObject`), whereas others must be used in certain cases for correct execution (`GetObjectContext.CreateInstance`, `Server.CreateObject`, and `New`). And some are more efficient than others, although one must take into account the type of activation, the interface being used, and the number of calls the client plans to make.

If you do not need COM activation, use `New` and vtable-binding. Otherwise, consult Table 2.2 to maximize performance. Although it may be counterintuitive, if your design involves "one-shot" objects (i.e., create, call, and destroy), then `CreateObject` with late-binding may be the most efficient approach. However, keep in mind that you lose IntelliSense and type checking with late-binding. For this reason, the conservative programmer should consider sticking with vtable-binding.

Rule 2-7: Beware of `Class_Terminate`

Two of the most heavily used classes in VB are probably `Connection` and `Recordset`, members of the ADO object model. Like most object-oriented classes, these two classes have destructors similar to `Class_Terminate`. In other words, they have methods that are automatically triggered when an instance of the class is about to be destroyed. Destructor methods are typically used to clean up before an object's state is lost forever, which would seem like the perfect place to handle things like saving changes to a database. So why is it, then, that `Connection` and `Recordset` have explicit `Close` methods that we have to call ourselves?

The answer is that some resources are too important to leave open until the client (or the run-time environment) gets around to triggering the object's destructor.[22] In other words, in VB, the destructor is triggered when an object is no longer referenced by any client:

```
Dim rs As ADODB.Recordset
Set rs = New ADODB.Recordset
  .
  .
  .
Set rs = Nothing   '** destructor is triggered at this point, assuming
                   '**    we didn't pass the reference to anyone else
```

This occurs when all references have been set to `Nothing`. For database classes like `Connection` and `Recordset`, which may be allocating memory and setting locks in the database, unnecessary delay in performing cleanup may waste precious resources and may hurt performance. Hence the explicit `Close` method:

```
Dim rs As ADODB.Recordset
Set rs = New ADODB.Recordset
rs.Open ...
  .
  .
  .
```

[22] The ideas in this rule will become even more important in .NET, which uses garbage collection and thus unpredictably delays object destruction.

```
rs.Close            '** cleanup performed here
Set rs = Nothing    '** destructor triggered here
```

Even though the client is responsible for calling `Close` (and thus may forget), its use can be documented as necessary for correct behavior. Regardless, it provides a solution to the problem of timely cleanup for those able to use it properly.

What does this mean to you? First of all, as a consumer of objects, you must be careful to use other classes properly. Look for methods entitled `Close` or `Dispose`, and be sure to call them as soon as you are done using that object. In particular, be careful to call these in your error handlers as well. For example, here's the proper way to ensure that both a connection and a recordset are closed, even in the presence of errors:

```
Public Sub SomeTask()
   On Error Goto errHandler
   Dim dbConn As ADODB.Connection
   Dim rs As ADODB.Recordset

   Set dbConn = New ADODB.Connection
   Set rs = New ADODB.Recordset
    .
    .
    .
   rs.Close : dbConn.Close
   Set rs = Nothing : Set dbConn = Nothing
   Exit Sub

errHandler:
   If rs Is Nothing Then
   Else
      If rs.State <> adStateClosed Then rs.Close
      Set rs = Nothing
   End If

   If dbConn Is Nothing Then
   Else
      If dbConn.State <> adStateClosed Then dbConn.Close
      Set dbConn = Nothing
```

```
    End If

    Err.Raise ...
End Sub
```

Second, as a producer of classes, you need to decide whether VB's `Class_Terminate` event is sufficient for your cleanup needs. If not, then you need to incorporate an explicit `Close` or `Dispose` method in your class design.

The current convention is to provide a `Close` method if your design allows an object to be reopened or reused after it has been closed. Otherwise, provide a `Dispose` method, which implies to your clients that the object is no longer usable once `Dispose` has been called. Implementing these methods is easy; the hard part is deciding when your classes need them.

The obvious examples are classes that open and hold on to operating system or other resources: files, shared memory, network connections, and ADO `Connection` and `Recordset` objects. If you find yourself opening these types of resources in your class's `Class_Initialize` event (or in an explicit `Open` method) and accessing them via private class variables, then you most likely need a `Close` or `Dispose` method. For example, you may design a data access class that automatically logs every access via a private ADO connection:

```
'** class module:  CDataAccess
Option Explicit

Private dbLog As ADODB.Connection   '** for logging accesses

Public Sub Open()
    Set dbLog = New ADODB.Connection
    dbLog.Open "<proper connection string>"
End Sub
  .
  .
  .
Public Sub Close()
    dbLog.Close
    Set dbLog = Nothing
End Sub
```

```
Private Sub Class_Terminate()
   Close   '** in case client forgets...
End Sub
```

Of course, keep in mind that although maintaining a dedicated logging connection may be efficient, it is certainly wasteful of an important (and usually limited) resource. In fact, we do not recommend the previous design for objects that may live in the middle tier, and thus need to scale (e.g., see rule 5-3). However, if such a design is appropriate in your case, then remember to think twice before relying solely on `Class_Terminate` for cleanup.

Rule 2-8: Model in Terms of Sessions Instead of Entities

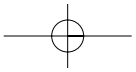
When it comes to designing the object model for your system, you should consider whether to design your classes around *sessions* or *entities*. Entities represent a more traditional object-oriented approach, in which classes are based on real-world entities in your system—customers, orders, products, and so forth. In contrast, sessions represent the set of expected interactions between clients and your objects. Although session-based class designs may deviate from pure OODs, the motivation is performance over elegance. Session-based systems strive to streamline client interactions, which is particularly important when objects are out-of-process (e.g., in distributed applications).[23]

Obviously, the issue of design is no small matter. For example, consider a multi-tier system with business and data layers. How should the data access layer behave?[24] Should it model each table as a class? If so, where do queries over multiple tables fit in? Perhaps there should be just one class per database. And what about the business layer? Are lots of smaller classes better than a few larger ones? How should they be grouped to take advantage of polymorphism in the client? Each of these questions may have different answers, based on system goals.

At a high level, most business systems are the same: They gather information from their users and submit this information for processing. The system is thus divided into at least two parts, the front-end user interface and the back-

[23] If you are new to out-of-process COM or why it is important, read rule 2-9 first.

[24] The interested reader should also see rule 5-2.



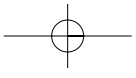
end processor. Each communication from the front-end to the back-end represents a unit of work and constitutes a *round-trip*. Session-based designs model user scenarios in an attempt to minimize round-trips. Traditional OODs often don't take into account the cost of a round-trip, yielding less than optimal performance.

For example, consider the traditional entity-based design of a `CCustomer` class. The class models the state and behavior of a customer in the system; in particular, allowing easy access to customer information:

```
'** class module:  CCustomer (traditional OOD)
Option Explicit

Public Name As String
Public StreetAddr As String
Public City As String
Public State As String
Public Zip As String

Public Property Get CreditLimit() As Currency
    '** return customer's credit limit for purchases
End Property

Public Sub PlaceOrder(ByVal lProductNum As Long, _
                      ByVal lQuantity As Long)
    '** code to place an order for this customer
End Sub
```

Although straightforward to understand, consider what the client must do to change a customer's address:

```
Dim rCust As CCustomer
Set rCust = ...

With rCust
    .StreetAddr = <new street address>
    .City = <new city>
    .State = <new state>
    .Zip = <new zipcode>
End With
```

The cost is four round-trips, one per datum. Likewise, placing an order for *N* different products requires *N* + 1 trips, one to check the customer's credit limit and another *N* to order each product.

A better approach, at least from the perspective of performance, is to redesign the `CCustomer` class based on the expected user scenarios: getting the customer's information, updating this information, and ordering products. This leads to the following session-based result:

```
'** class module:  CCustomer (revised session-based design)
Option Explicit

Private Name As String          '** no public access to data
Private StreetAddr As String
Private City As String
Private State As String
Private Zip As String

Public Sub GetInfo(Optional ByRef sName As String, _
            Optional ByRef sStreetAddr As String, _
            Optional ByRef sCity As String, _
            Optional ByRef sState As String, _
            Optional ByRef sZip As String)
   sName = Name
   sStreetAddr = StreetAddr
   sCity = City
   sState = State
   sZip = Zip
End Sub

Public Sub Update(Optional ByVal sName As String = "?", _
            Optional ByVal sStreetAddr As String = "?", _
            Optional ByVal sCity As String = "?", _
            Optional ByVal sState As String = "?", _
            Optional ByVal sZip As String = "?")
   If sName <> "?" Then Name = sName
   If sStreetAddr <> "?" Then StreetAddr = sStreetAddr
   If sCity <> "?" Then City = sCity
   If sState <> "?" Then State = sState
   If sZip <> "?" Then Zip = sZip
End Sub
```

```
Public Sub PlaceOrder(laProducts() As Long)
    '** confirm that client passed a 2D array (products times quantities)
    Debug.Assert UBound(laProducts, 1) = _
                 UBound(laProducts, 2)

    '** code to check that credit limit is sufficient
    '** code to place entire order for this customer
End Sub
```

First of all, notice there is no public access to customer data. All reads and writes must be done via methods. As a result, an address change now takes only one round-trip call to `Update`. Likewise, `PlaceOrder` is redesigned to accept an array of product numbers and quantities, allowing an entire order to be placed via one round-trip call. In short, the class contains one entry for each task that the user may need to perform. Although the class's interface is arguably more cumbersome for clients to use, the potential increase in performance is significant, especially across a network.

Session-based designs are usable at every level of a system. For example, in a standard multi-tier application, your business objects would model client sessions, whereas your data access objects model business object sessions. For the latter, your data access design may be as simple, and as efficient, as two methods: one to read and one to write:

```
'** class module:  CDataAccess (minimal session-based design)
Option Explicit

Public Function ReadDB(sConnectionInfo As String, _
                       sSQL As String) As ADODB.Recordset
    '** code to open DB, build recordset, disconnect, and return it...
End Function

Public Function UpdateDB(sConnectionInfo As String, _
                         sSQL As String)
    '** code to open DB and update via SQL...
End Function
```

Obviously, good design is the proper balance of usability, maintainability, extensibility, and performance.

Rule 2-9: Avoid ActiveX EXEs Except for Simple, Small-Scale Needs

When you create a new project in VB, you are presented with a list of project types from which to choose: Standard EXE, ActiveX EXE, ActiveX DLL, ActiveX Control, Addin, IIS Application, and so forth. The ActiveX project types are used when you want to create a COM server—a set of classes in a DLL or EXE that can be activated using COM. If your goal is a user-interface component, select ActiveX Control. However, if your goal is a traditional object-oriented, non-user interface component, then you should select either ActiveX DLL or ActiveX EXE. But which one?

The answer depends on two factors: the type of COM activation you desire, and whether you plan to use MTS or COM+. Let's review the three types of COM activation: *in-process*, *local*, and *remote*. An in-process activation means the COM object resides in the same process as the client that created the object. In this case, you must create an ActiveX DLL project, and the resulting DLL must be installed on the client's machine. Both local and remote COM activation represent *out-of-process* activation, in which the object resides in a process separate from the client—on either the same machine (local) or a different one (remote). With this scenario you have a choice. You can create an ActiveX EXE project, and the resulting EXE serves as a stand-alone process for hosting your objects, or you can create an ActiveX DLL and configure it to run within MTS or COM+ as a server process.

In-process objects are much more efficient, because calls are typically 10 to 100 times faster than calls out of process. The trade-off is that out-of-process objects offer

- Fault isolation (object can crash without crashing the client, and vice versa)
- Separate security identity (object runs under an identity separate from the client)
- Multi-threaded behavior (clients can concurrently activate objects/execute calls)
- The ability to run objects on a machine separate from the clients

The last is perhaps the most important, because it enables the construction of distributed, multi-tier applications. Assuming you want out-of-process activation, the question is should you use ActiveX EXEs or should you turn to MTS/COM+?

In short, VB's ActiveX EXEs are designed to support small-scale needs. They provide basic out-of-process activation, nothing more. On the other hand, MTS and COM+ support large-scale designs, in addition to providing a host of other services: security, resource sharing, distributed transactions, and configuration/process management. When in doubt, the general consensus is to use MTS or COM+, because you never know when you may need to handle additional clients, share resources among your objects, or implement security. However, if your needs are simple, then VB's ActiveX EXEs are a viable option. Because Chapter 3 focuses entirely on MTS and COM+, we discuss ActiveX EXEs here.

VB's ActiveX EXEs enable you to build multi-threaded server applications with relative ease. Like many features of VB, multi-threading is presented through the IDE with the utmost consideration for productivity. In this case, your ActiveX EXE's threading strategy is determined by two option buttons and a text box, not by coding. These Threading Model settings are found in your project's properties, under the General tab as shown in Figure 2.20.

An ActiveX EXE is compiled to follow one of three threading model approaches. The default is Thread Pool of 1 (shown in Figure 2.20), which gives you a single-threaded application. This means that a single-thread is shared by all objects living in this server process, and thus only one client request can be processed at a time. Although this type of server consumes very few resources, it should be used only when you are supporting a single client.

The second approach is Thread per Object, which represents the other end of the threading spectrum. Now, instead of one thread, every object has its own thread.[25] The result is maximum concurrency, because no client request blocks that of another. However, even though the server process supports an unlimited number of concurrent clients, does it maximize throughput? Not likely. At some

[25] Assuming the object was activated by a client outside the server process. If the client lives inside the EXE, then the new object lives on the same thread for better performance. Note that setting a class's instancing property to `SingleUse` has a similar effect—an entirely new EXE is started for each instance of that class.

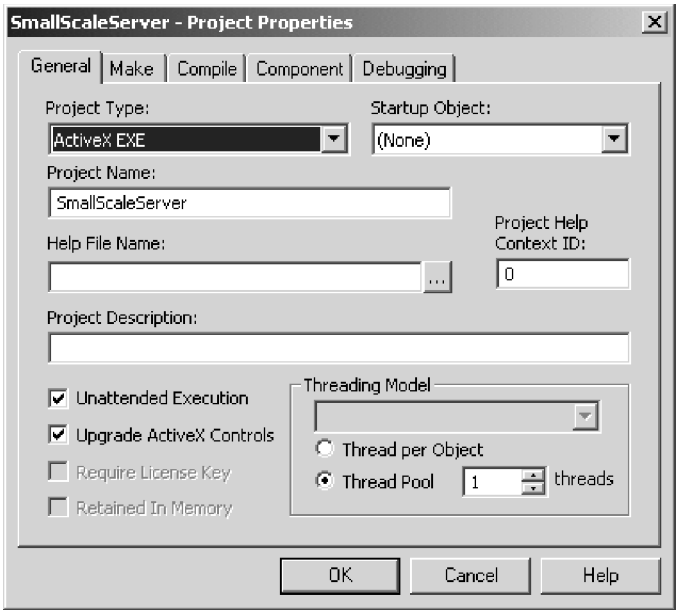


Figure 2.20 Project properties for an ActiveX EXE

point, the rising number of threads begins to hurt performance, because the operating system spends more time switching from one thread to another than it does letting a thread run. Thus, if you want to maximize both concurrency and throughput, you need to either *scale up* (add more hardware to the existing machine) or *scale out* (add more machines and load balance). You'll also need to cap the size of the thread pool—the motivation for the third approach.

The third (and best) approach for multi-threading is a thread pool more than 1. The idea is to limit concurrency by restricting the number of threads, thereby guaranteeing some base amount of throughput as the load on your server process increases. For example, Figure 2.21 shows a VB ActiveX EXE compiled with a thread pool of 3. The threads are depicted as circles with arrowheads, and each thread is assigned to a single *apartment* within the process (hence the term single-threaded apartment, or STA). When VB objects are created, they are likewise assigned to an apartment, and remain in that apartment until they are destroyed. Although the ActiveX EXE can support an unlimited

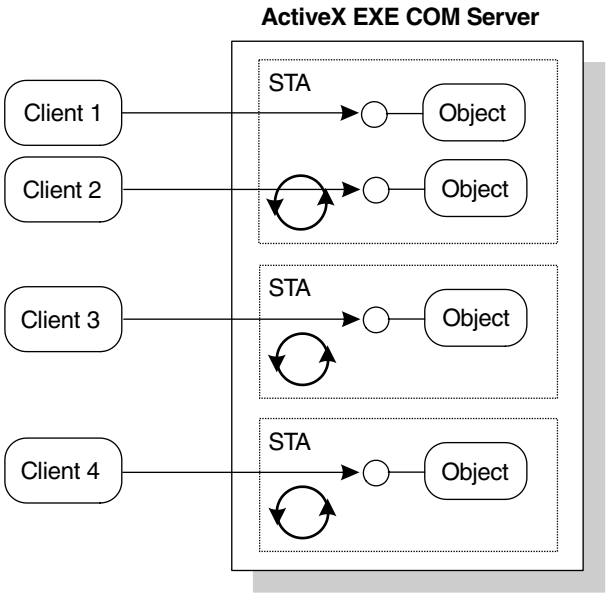


Figure 2.21 ActiveX EXE server with four clients and a thread pool of 3

number of objects (and hence an unlimited number of clients), in this example only three client requests can be processed concurrently. In particular, note that clients 1 and 2 have objects assigned to the same thread. Thus, if both submit a request (i.e., make a method call) at the same time, one request is processed and the other is blocked.

The idea of a fixed-size thread pool is not unique to VB. MTS and COM+ also take this approach: MTS 2.0 on WinNT has a thread pool of 100, whereas COM+ (i.e., MTS 3.0) on Windows 2000 has a thread pool per processor ranging in size from 7 to 10. Notice that the size of the pool was reduced significantly in COM+, acknowledging the tension between concurrency and throughput.

Even though their thread pooling strategies are the same, note that MTS and COM+ provide a more sophisticated implementation, yielding better performance than VB's ActiveX EXEs. MTS and COM+ provide other distinct advantages as well. For example, consider the problem of objects trying to share

data such as configuration information or a set of database records. In VB, the standard approach is to use global variables declared in a BAS module. However, in an ActiveX EXE, global variables are not truly global: A BAS module is replicated so that each apartment has its own copy. The result is that "global" variables are global only within an apartment.[26] This implies that you must use an alternative mechanism to share state, such as a file or database, or the memory-based Shared Property Manager within MTS and COM+.

With regard to security, VB's ActiveX EXEs rely on COM's security model. Using the `dcomcnfg` utility, you can configure the identity under which an ActiveX EXE runs, as well as who may start up, access, and configure the EXE. This also applies to the authentication level (frequency of authentication and network packet integrity/privacy).

In summary, ActiveX EXEs provide a quick-and-dirty mechanism for out-of-process COM activation, and thus are a basis for application designs requiring fault isolation, security, concurrency, or distributed processing. However, keep in mind that Microsoft is moving away from ActiveX EXEs, and is encouraging developers to build ActiveX DLLs and to let MTS or COM+ serve as your EXE. This allows Microsoft to provide services that are difficult to implement yourself, and the ability to evolve these services without the need for you to recompile your code. Applications based on MTS and COM+ will scale, offer better concurrency and resource sharing, allow more flexible configuration of the server, and yield faster time-to-market for multi-tier systems. In the end, you'll spend your time more productively, working on business logic rather than infrastructure.

[26] Why did the designers of VB do this? Consider the alternative: If global variables were shared across apartments, then programmers would need to worry about synchronization—a slippery slope that leads to subtle, error-prone code. Another side effect: If you start up via a `Sub Main`, it is run *each* time a new apartment is created.