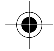
Chapter 10

Building Custom Commerce Components

by Biff Gaut

This chapter will provide some very useful tips for designing COM objects providing Web services. But, in order to successfully augment your Active Server Pages with COM, it is important to follow seven very basic rules. If you do, then you will avoid the most common mistakes encountered when programming on a grander scale. First you will hear about two fictional Web sites that make some fundamental mistakes in their development and get a taste of the dangers that lie in wait for careless COM developers. After hearing their plight, you will find seven simple rules that would have helped them and definitely will help you.

Active Server Pages (ASP) is definitely the way to begin programming a Web application. It lets you create dynamic content on a Web server in a very straightforward way. Why, then, is it only the start? Why shouldn't all Web content be developed with ASP? If you've read Chapter 7, you know that ASP is easy to write and provides decent performance. Before you start doing the



final design of your new site, however, let's look at two examples where you might need to start thinking about the next step.

Reuse, Reuse, Reuse

The Web site at *www.webofdeceit.com* is home to a brand new Internet startup company peddling mystery novels over the Web. (Don't bother looking. I'm making this up.) They've assembled a vast database of mystery trivia in a comprehensive database with access by an ASP application. Now your aunt can go to the Web and find out that Jimmy Soames was killed by a pathological unicycle rider in *A Spoke Too Soon*. This application then allows, even encourages, a user to jump directly to the storefront page to place an order for the book. Fortunately, it seems that *www.webofdeceit.com* strikes a nerve, and everyone who ever read a mystery decides to go out and try the database and buy a book. Overwhelmed by their good fortune, the owners decide to open up a storefront to start selling books over the counter as well as over the Web. A customer can come into the store and ask who was the serial killer of street performers in *A Mime Is a Terrible Thing to Waste*. The clerks figure that because the machine with the search application is connected to their workstations by a Local Area Network, they should be able to get a little better performance and a little more functionality than someone coming over the Web. Oops. All the logic is in ASP pages. ASP pages can operate only under the auspices of the Internet Information Server (IIS), so the clerks have to log on to the Web site like everybody else. A better design would locate the logic where the ASP pages can still use it, but other applications can use it as well without having to go through IIS. This is the first big argument for COM. Consider that this site will also rely heavily on Site Server Commerce services (see Chapter 6) with all of its COM object needs, and you should be just about convinced that writing COM objects is in your future. Just in case you're still skeptical, here's one more example.

Developing on a Grand Scale

A home improvement company offers a Web site with the capability to enter the dimensions of your kitchen, along with options as to your decorating desires (cabinet style and location, appliances, and so on) to produce a three-dimensional rendering of what your new kitchen will look like. Unfortunately, these extremely complex calculations take a long time to run, especially written in interpreted ASP. Though the site designers are not worried about making the would-be redecorator wait several seconds to see the results, they are concerned that a couple of users browsing for a new kitchen could slow their

server to the point where a potential customer can't log on to the site to get the phone number of the firm. Their need is to write the rendering code in a faster language and run it on a different machine. Both these goals can be met by COM as well. COM will allow functionality to be written and shared in any language and, through distributed COM, allow the processing load to be shared across many machines.

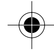
The point of these two examples (and many other situations that aren't listed here) is that ASP can take you only so far. If you are creating a large-scale Web site, you will soon find yourself at a point where you can't add any more processors to your Web server, and your calculations are taking too long to run in ASP. Make no mistake, ASP is instrumental to the development of any Web site, large or small. The best way to use ASP, however, is as a scripting language to control your COM components. Avoid the situation described in these examples by planning for them up front and putting your code in reusable COM objects. Other benefits you'll see from using COM objects to implement your business rules are:

- The object-oriented nature of COM will allow more compartmentalized development. This will enable more developers to work together and reduce the likelihood of dreaded spaghetti code.
- COM objects allow you to take advantage of the scaling and transaction features of Microsoft Transaction Server, described in Chapter 11.

COM SERVERS FOR THE WEB

Hopefully, by this time you're convinced that it's important to move functionality into COM components to make your Web site run. But what type of functionality is appropriate to implement with a COM component? Just about anything. If you have been programming in a Microsoft desktop environment, you have probably already used many COM components, maybe without even knowing it. A number of places where you might have used a COM object are:

- ActiveX Controls in Microsoft Visual Basic. ActiveX Controls are simply COM objects that expose a set of interfaces defined for ActiveX Controls[1].
- Programming Microsoft Office, either from outside using Visual Basic or from within using VBA. The object hierarchies exposed by the Office Applications are really a set of COM objects.



- Writing a Visual Basic class. A Visual Basic class, in reality, is a COM object.

Making your own COM objects is a relatively simple and straightforward process. Microsoft has supplied wizards in all of their major languages to simplify COM programming. Chapter 12 demonstrates the basics of creating a COM object in Visual Basic, Microsoft Visual C++, and Microsoft Visual J++.

Although almost any piece of code can be implemented as a COM object and any COM object can be run on a server, the demands on a server-side component require special consideration. For instance, although Microsoft Excel exposes almost all of its functionality through a hierarchy of COM interfaces, several features discussed later in this chapter make it a poor choice for most server-side functions.

SEVEN RULES OF COM DEVELOPMENT

If you follow a few rules when you develop your COM components, your site will be faster, easier to support, and less likely to crash. Learning these rules in advance can save you hours of heartbreak and lost sleep in the last few weeks before delivery.

Rule #1: Server-Side COM Objects Have No User Interface

The first rule when designing a COM object to run on the server is that the object is not allowed to have *any* user interface. Think about this for a minute. These objects will be running on the server where they will be created and manipulated by IIS. IIS runs as an NT Service, so it, along with the objects it instantiates, has no access to the desktop. If an object attempts to display a dialog box asking for additional user input or displaying information, it will fail. If the UI requires input before processing can continue, the http request will freeze. Although the client side will eventually time out, the request will remain frozen on the server side. Because IIS has the capacity to service a limited number of http requests simultaneously, every time one request gets halted, the overall response of your site will worsen. Freeze enough sessions, and IIS will halt altogether. Sooner or later, this process will end with you or the System Administrator getting awakened in the middle of the night to go down to the office to restart IIS so your site can resume business.

Because we already determined that all user interface functionality should be handled by the browser using ASP-generated HTML and DHTML, this rule might seem to be redundant. But obeying this rule might not be as straightfor-

ward as you first think. Your object must have *no* user interface, not even something stuck in by the compiler itself. For instance, in VB any uncaught error will throw a dialog box up on the screen (see Figure 10-1). Even though you did not put a dialog box in your code, the IIS session is frozen. Similar things can happen in VC as well. When implementing your object, you have to be sure that you have planned for these contingencies. In Visual Basic, go to the Project Properties dialog box and check the Unattended Execution box (see Figure 10-2), and write error handling code in every routine to catch exceptions. In Visual C++, make sure that you catch all exceptions by setting up a try/catch block at the beginning and end of each interface method.

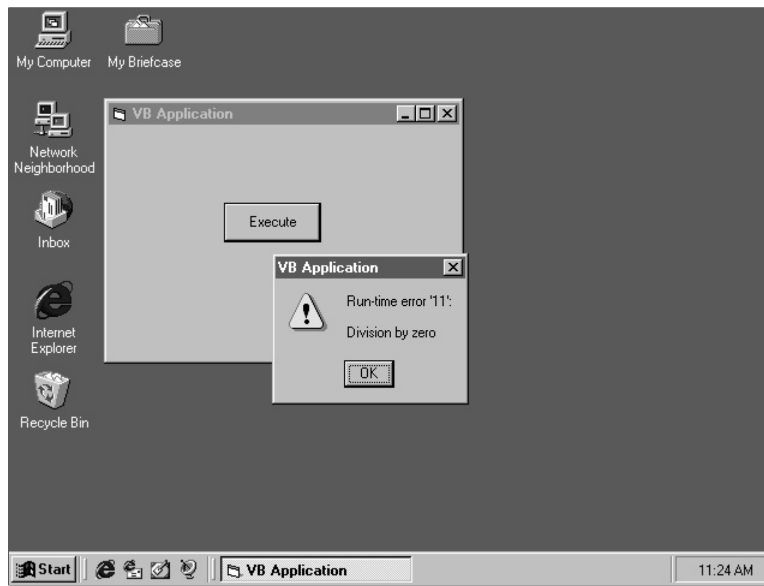


Figure 10-1. *Did you see this one coming? An error that is bothersome in Visual Basic is disastrous on a Web site.*

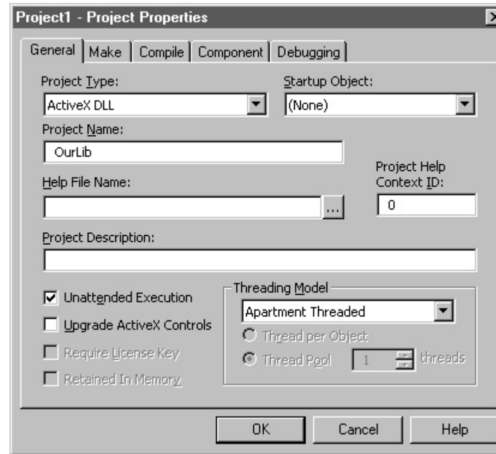


Figure 10-2. *The Unattended Execution feature will prevent unwanted user interface.*

The other place to watch out for unexpected UI is inside existing, or legacy, code. One common use of COM is to wrap legacy code in COM objects to provide its functionality to a whole new set of clients and thereby extend its life. Make sure you scour that old code well for any hint of a user interface before you put it on your server or some long-forgotten dialog box in a low-level function will slow your site to a crawl. So, although this seems like a straightforward rule, it's important not to overlook anything, or you'll inconvenience your customers and annoy your coworkers.

Rule #2: Report User Errors to the User and System Errors to the System

Two things can go wrong while a user is using your Web site. First, the user might enter bad information. If this happens, your COM objects should catch the error and, in conjunction with the ASP code, inform the user of the mistake in as helpful a manner as possible. These are normal errors, and programming for them on the Web is very similar to programming for them on the desktop.

The other type of error is a system error. This might include things like a database crash or a disk full error. Although these are errors, they must be handled quite differently from user error. How many times have you been using a Web site and instead of a page you were expecting, you got a message like the one in Figure 10-3?

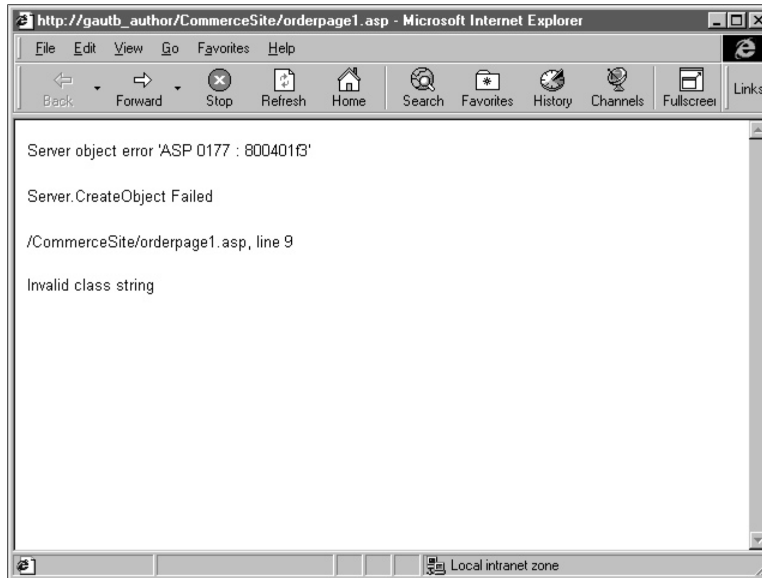


Figure 10-3. *Reporting a system error to the user.*

This is a situation where you, as a developer, must be intelligent in your error handling and, when the site has problems, inform the user in a clear and non-threatening way. If a site was down, wouldn't you rather see a message more like the one in Figure 10-4?

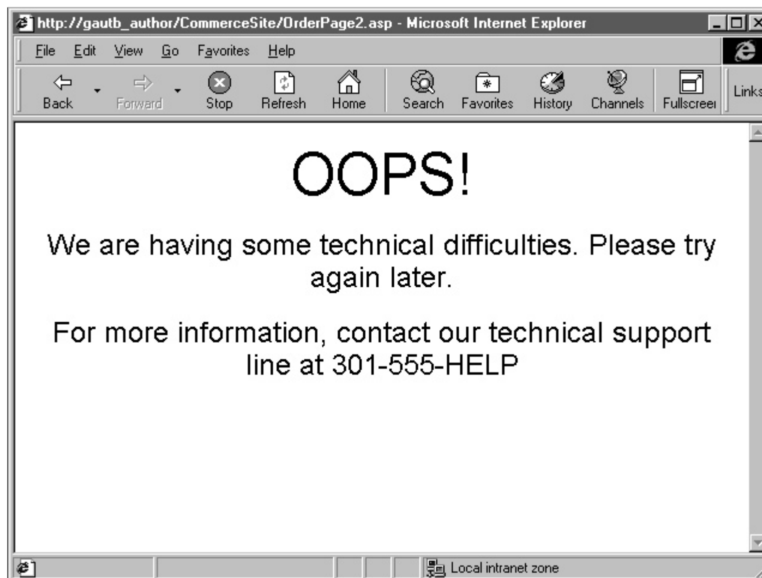


Figure 10-4. *Hiding system error information from the client.*

Unfortunately, if your code encounters a system error and you shield the user from the details, your job is only half done. You must record the error somewhere and notify someone that the error has occurred. Appropriate places to log errors include the NT Event Log (pictured in Figure 10-5), a SQL Server database, or a simple disk file. In Visual Basic, putting an entry in the event log is as easy as calling the *LogEvent* method on the *App* object. To notify the System Administrator, your options are limited only to how creative you are and how much time you have. One relatively simple option is to use MAPI (another COM interface!) to create an e-mail message to the System Administrator detailing the problem. This message can then be sent to multiple locations, including mail addresses acting as a front for a pager.

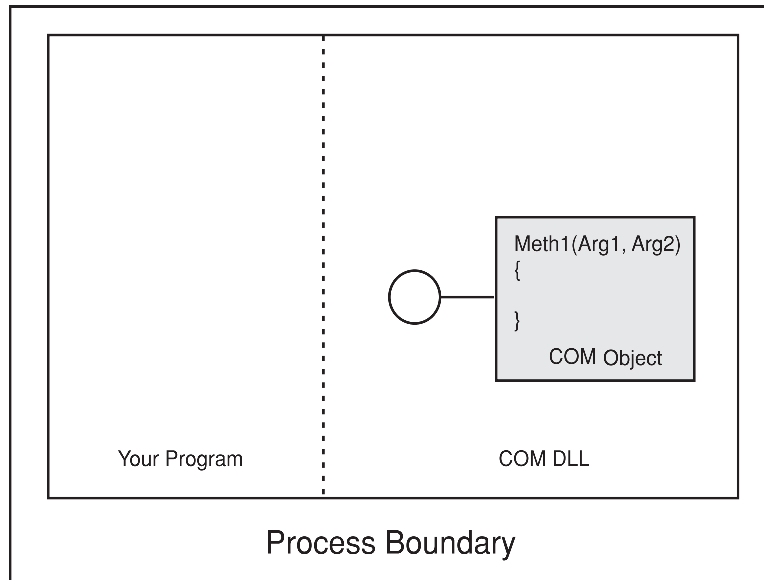


Figure 10-5. *The Windows NT Event Log.*

Rule #3: If At All Possible, Server Objects Must Be in Process

COM objects can be implemented in two ways: In Process and Local. In Process objects are implemented in a dynamic link library (DLL) and run in the same process space as the client program. In this way, they essentially become a part of the client program when loaded (see Figure 10-6). Local objects are implemented as executable programs that run in their own process space. When they are created, the object's executable program is loaded and runs alongside the client program (see Figure 10-7). Although this doesn't seem to be a significant

difference, the difference in performance between the two implementations can be substantial.

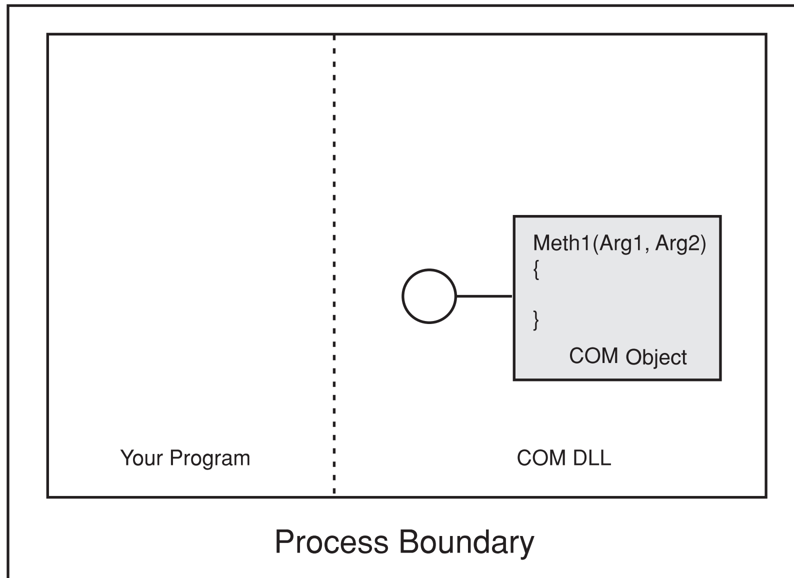


Figure 10-6. *An In Process COM object.*

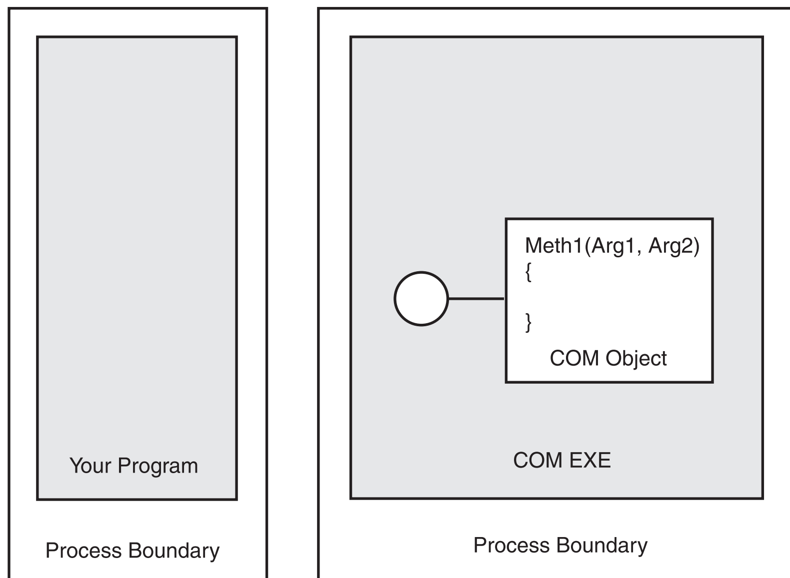


Figure 10-7. *A Local COM object.*

Figure 10-8 shows a client calling a method on an In Process object. When the method is called, the arguments are packaged up into an area of memory called the stack. The In Process object can access them directly. Compare this with Figure 10-9, which shows a client calling a method on a Local object. When the method is called, the arguments are again packaged up into the stack, but because the object is in a different process that has its own stack, it cannot access the stack that contains the arguments. COM steps in to help at this point. COM automatically takes the arguments, converts them to a buffer of raw binary information, and transfers them over to the process containing the COM object and reassembles them in the stack for the COM object to access. Once the method is complete, the return values are treated the same way to get back to the client. This system of automatically moving arguments and return values from one process to another is called *marshalling*, and although it happens automatically, it can be very expensive from a performance perspective.

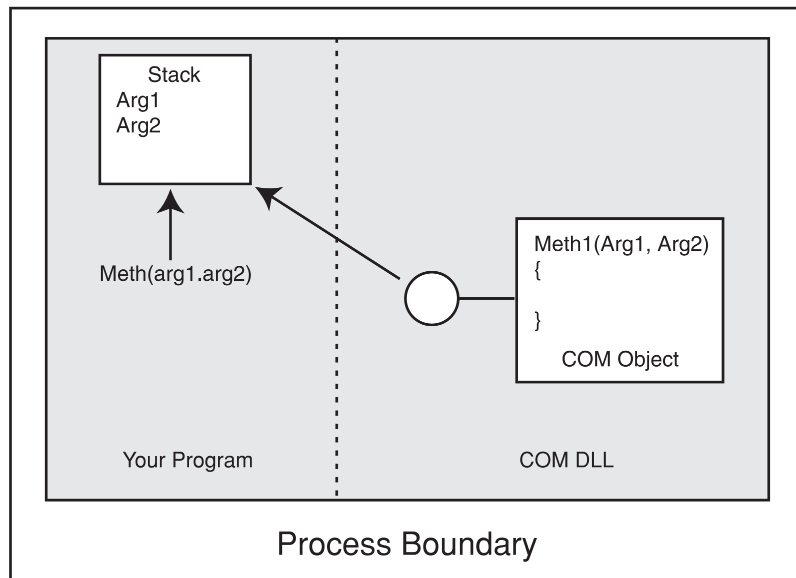


Figure 10-8. *Sharing arguments with an In Process server object.*

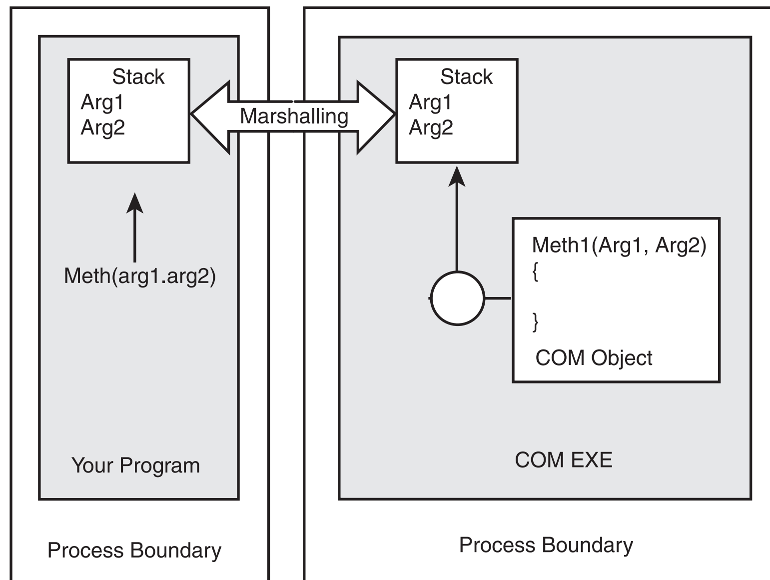


Figure 10-9. *Sharing arguments in a Local server object.*

Just how expensive is marshalling? I have run tests that do nothing but execute extensive COM calls and have found that the Local versions of objects have performed up to 5,000 times slower than the In Process test. These tests, however, didn't do anything other than make calls to empty methods. In a real-world application, the code implemented within a method would take much longer to execute than the duration of the marshalling, thereby lessening marshalling's influence on the overall performance of the object.

Rule #4: Server-Side Objects Must Be As Small As Possible

This rule applies to almost all programming situations, but it is brought up here to highlight some special aspects of server programming that make it all the more important. Let's say you have a very sophisticated financial model implemented in an Excel worksheet that you want to make available to your users over the Web. Because Excel is an automation server, it is possible, in fact, it is pretty simple for you to write ASP code that instantiates Excel, loads the worksheet, runs the spreadsheet model, and returns the result. Let's look at two situations that suggest that this is not such a good idea.

The first user of the day logs onto your site and runs the financial model. In response to that request, you are going to load Excel and load a spreadsheet. Do that once by hand on your computer and see how long it takes. Considering that a responsive Web site generates its pages in well under a second on

the server side, imagine how responsive your site is going to seem to your user. One of the reasons these objects must be very small is so that load time is very short.

Later in the day, as more users arrive at the office, they all start logging on and requesting models. Even if you have set up your system so that one instance of Excel can service many requests, you're still creating a new workbook object for each user. If you have hundreds of users logged on at the same time, you're going to have a lot of memory on your server taken up by Excel. Remember that any object you use could conceivably get instantiated hundreds of times—another good reason to keep them small.

Don't let this section make you think that there is no place on a Web server for Excel—quite the contrary. Excel is a wonderful tool for Web applications; it just needs to be used correctly. We'll take a look at an application that uses Excel appropriately in Chapter 17.

Rule #5: Handle Errors According to COM Conventions

A key aspect of developing in any type of environment is error handling. Whether the error is system generated or based on invalid user input, a development team must decide how they are going to notify each other of errors and then stick to that standard (hopefully!) across the entire project. Developers working on a COM object or a hierarchy of COM objects is no different. There are many ways to get error information from a COM object back to your clients. For instance, you can set up a system of return values or use an attribute of each object that defines error status. One key point that slips by many new COM developers is that COM has already defined a standard method to notify clients of errors. It is in your best interest to learn and implement COM's standard for two reasons:

- In the general sense, because COM is set up as an industry standard for component software, clients that have no advance knowledge of your component will be able to read your errors without changing their code.
- From a more pragmatic point of view, Visual Basic, Visual C++, and Visual J++ automatically support the standard COM error system when writing both clients and components, making it very easy to implement the COM standard. ASP also looks for the COM error standard.

So, besides being the proverbial "right thing to do," using the COM error mechanism is easier than defining your own. Let's talk about how it works, first from a detailed point of view, and then from a more practical point of view.

COM errors

Although the Microsoft Visual Studio languages all have features that make using the COM error mechanism relatively simple, it is best that you have a reasonable understanding about what is going on under the hood as well. This will make things much clearer when you start developing your objects in multiple languages.

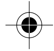
In COM, an error is associated with a concept called a "logical thread." A logical thread is a series of nested method calls, similar to a call stack in a single language. The difference between a logical thread and an operating system thread (to be discussed shortly) is that a logical thread may actually span more than one operating system thread.

When an error occurs in a COM method, an object following the COM standard will handle the error by creating an object that supports the IErrorInfo interface. An object that implements the IErrorInfo interface is referred to as an ErrorInfo object. This interface provides a large amount of information about the error, including:

- The interface where the error occurred
- The text name of the class where the error occurred
- A textual description of the error
- The filename of the Windows help file with the full documentation of the error
- A help context into the Windows help file

The object might either implement its own ErrorInfo object, or it might use a default implementation exposed by the COM libraries. Needless to say, the default implementation is a lot more popular. Once this object has been created, a COM API call allows the method to associate with the logical thread. The method now returns an HRESULT[2] that indicates to the caller that an error has occurred. It is important to note that the ErrorInfo object will then remain associated with the logical thread until some method higher up the chain explicitly removes it with another COM API call.

Of course, putting an ErrorInfo object on the logical thread doesn't do any good unless the client knows it is there. An object must have some way to alert a client that it is playing by COM's rules. It's probably not a surprise that COM has this covered as well. An object that supports COM error handling should



implement the ISupportErrorInfo interface. This interface provides a method (InterfaceSupportsErrorInfo) that will tell the client whether or not an ErrorInfo object can be expected from this interface for this object. This concept of telling the client that you are playing by the rules illuminates another reason that you should follow the rules yourself. Suppose you are making use of a third-party object inside one of your methods. This third-party object is a good citizen, so when it has an error, it creates an ErrorInfo object and informs you that the ErrorInfo object is out there. You, on the other hand, did not bother implementing COM error handling, so when you handle the third-party object's error by returning an error HRESULT to your caller, you also tell the caller that you do not support the ErrorInfo object paradigm. Your caller, on hearing this, doesn't bother looking for an ErrorInfo object, even though one exists for the logical thread! By not playing by the rules, you not only make your errors harder to deal with, you also hide any errors that occur within servers you use from your clients.

Visual Studio to the rescue

Although the preceding example makes the concept of ErrorInfo objects seem like a lot of work, thanks to some wizards and functions in Visual Studio, they are actually quite easy to deal with. Here is a quick summary of each language's support for ErrorInfo objects.

Visual C++

When you write an ATL object, the Error method of the CComCoClass will take a large variety of arguments and create the ErrorInfo object automatically. There is also a check box on the Create Object Wizard that will add an implementation of the ISupportErrorInfo interface to your object.

Using the Visual C++ #import directive to import a COM library will create a set of smart pointers for that library's COM objects. In addition to providing simple access to the object's properties and methods, these smart pointers provide a wrapper function around every method. These wrapper functions check the HRESULT returned from the wrapped COM method and, in the event of an error, retrieve the ErrorInfo object and convert it to a C++ exception.

Visual Basic

When writing an object in Visual Basic, you can create an ErrorInfo object by using the Visual Basic Err object. The Raise method of the Err object takes all the information required to create the ErrorInfo object as arguments. Inside the Raise method, the object is created and associated with the logical thread. Any Visual Basic objects you write automatically implement the ISupportErrorInfo with no special action required on your part.

If you're a Visual Basic developer, all this talk of COM methods returning

only HRESULTs has probably left you confused. From a Visual Basic perspective, COM methods return numbers, strings, and so on without an HRESULT in sight. This is because when acting as a client, Visual Basic always provides a layer of abstraction around COM calls that prevents the developer from having to worry about HRESULTs. When Visual Basic intercepts a bad HRESULT, it gets the ErrorInfo object and converts it to the Visual Basic Err object and throws a Visual Basic error that can be caught with an On Error Goto statement.

Visual J++

Visual J++ uses the ComFailException class to translate between Java exceptions and ErrorInfo objects. When a COM server encounters an error, it will create and throw a ComFailException object that contains all the information required to create an ErrorInfo object. The ComFailException is found in the com.ms.com package.

When Visual J++ acts as a COM client, it still uses ComFailException to handle COM errors. In this case, the ErrorInfo object is retrieved, converted to a ComFailException object, and thrown.

Rule #6: An Object Might Not Have the Privileges You Expect

Suppose you have written an object that, as a part of its responsibilities, reads some configuration data from the registry. As a careful developer, you've also written a test program in Visual Basic that thoroughly exercises all the methods and attributes of the object. You've run stress tests on the object to test for memory leaks. You've finally declared the object to be complete and robust and have loaded it on your Web server. Unfortunately, the first time you used the object from your ASP code, the exact same code that you used in Visual Basic, the object failed miserably. You have just learned your first lesson in Windows NT security.

When you run a Visual Basic program, the program and any objects it instantiates have the same privileges as the currently logged on user. Being a developer, chances are pretty good that you have Administrator privileges on your local machine, and your programs are the baddest dudes in the land, strutting around your machine unmolested by the security issues that may hinder lesser programs. ASP runs under the auspices of an NT service, however, so it doesn't even require a user to be logged onto the computer to execute. Instead, it operates under the auspices of a special username, IUSR_*machinename*. This username, usually called the anonymous user account, is given only guest privileges to the resources of the computer. It is possible to increase the privileges of the anonymous user account, but because many IIS machines are outside the

company firewall, it is probably not a good idea. Limited to the usual guest privileges, your objects may not have the privileges to do some of the things that you expect. Like all guests, however, your objects will have the right to hog the remote, eat all your food, and stick around too long.

The bug in the first paragraph is based on a true story. It seems the object was attempting to open the registry in read/write mode when read only mode would have been sufficient. This was fine when the object had the user's Administrator rights but impossible for a guest account. Changing the open mode of the registry to read only solved the problem, and everything turned out fine. There are a lot more details to NT Security, but just knowing this rule should steer you clear of your first few potential problems.

Rule #7: Learn the Relationship Between Object Scope and Threads of Execution

Did you know that there are certain instances where it is completely inappropriate to use a Visual Basic object on your server? If you do not follow Rule #7, you may fall into this trap. This is probably the most complicated rule in this chapter, but ignoring the relationship between object scope and threads can have a huge impact on your site's performance. To understand this relationship, you first need to understand what each of these terms means.

Object scope

An object's scope defines where it can be accessed in a Web application. This availability is determined by where the object is created and how it is stored. In a Web application, there are three possible scopes for an object:

- **Application Scope.** Objects stored in the ASP Application variable have application scope. Unless they are explicitly removed from the Application variable, they will exist as long as IIS is running the Web application where they are housed. Any http request by any user can access them, and every one of the requests will use the exact same instance of the object.

```
<%
' This ASP code creates and stores an
' application scope object
Set Application("AppObj") =
Server.CreateObject("WebApp.MyObject")
%>
```

- **Session Scope.** Objects stored in the Session variable have session scope. Like objects with Application scope, unless they are explicitly

removed from the Session variable, they will exist until the session (one user's interactions with the site) is completed. All the http requests from a user share the same session scope objects, but different users cannot share objects stored at the session level.

```
<%
' This ASP code creates and stores a
' session scope object
Set Session("SessionObj") =
Server.CreateObject("WebApp.MyObject")
%>
```

- **Request scope.** Objects created, utilized, and freed during the course of one http call have request scope. These objects exist strictly for the duration of the call and cannot be shared between requests, regardless of which user is making the requests.

```
<%
' This ASP code creates and stores a
' request scope object
Set Obj = Server.CreateObject("WebApp.MyObject")
%>
```

As you will soon learn, the scope of an object determines a great deal about how the object is accessed and, therefore, how it must be written.

Threads

Let's go back to Computer Science 101 for just a moment. A computer program is merely a list of instructions executed in order. The process of executing these steps is known as a "thread of execution." As long as only one step is being executed at a time, the program is single threaded. If the process of executing the steps in a program is going on in several parts simultaneously—that is, if something is executing Step 2 at the same time something else is executing Step 4—then the program is multi-threaded. This is a very hard concept to visualize the first time. Let's use a real-world example to make it clearer.

Think of your last trip to the department of motor vehicles to get your driver's license. When you arrive, there is a sign with a list of steps you must take to get your license renewed. These steps might include:

1. Go to station 1 and pick up a form to fill out.
2. Take the completed form to station 2 to pay your fee.
3. Take your receipt and form to station 3 to get your picture taken.
4. Go to station 4 to await your completed license.

This set of steps is illustrated in Figure 10-10. The dotted line indicates your path around the stations.

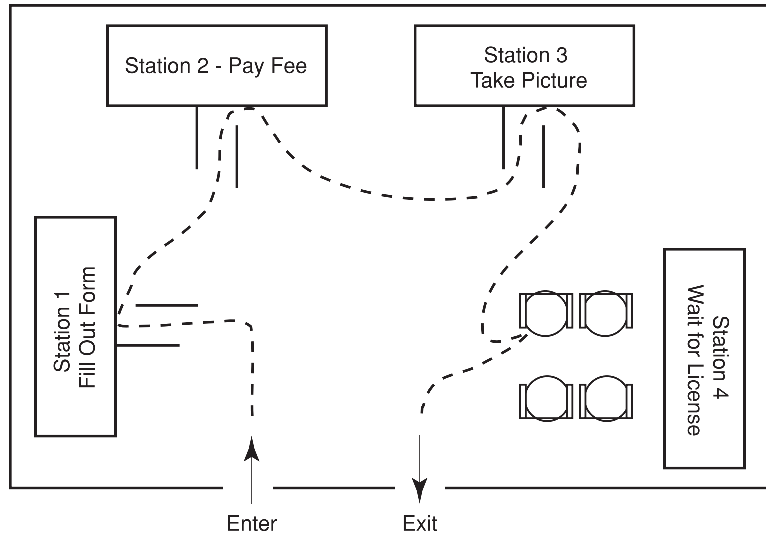


Figure 10-10. *Executing the steps of the "Get Your Driver's License" program.*

If you think of the steps of getting a driver's license as a computer program, your path through those steps is a thread of execution. If only one person were allowed in the department at a time, getting your license would be a single-threaded process. A computer program works the same way: If only one instruction can be executed at a time, the process is single threaded. In reality, the concept of only one person going through the process of obtaining a driver's license would be very wasteful, because three stations would be sitting idle for long periods of time. In fact, it would take even longer to get a driver's license than it does now!

It's much more efficient to have many people executing the steps to get their licenses simultaneously, with each one either performing the function of a station or waiting in line for their turn to use the resource found at a station. Getting your license is then a multi-threaded process, as illustrated in Figure 10-11. Each person can be thought of as a thread of execution, with each thread executing a different function simultaneously. A multi-threaded computer program works the same way, with many threads executing instructions in different parts of the code at the same time. When two threads of execution require the same resource—for example, the same block of memory, there are special programming constructs to make sure that one thread waits patiently for the other to finish using the resource before beginning. This is similar to how patrons of the DMV wait patiently for their turn to have their picture butchered

by the camera at station 3!

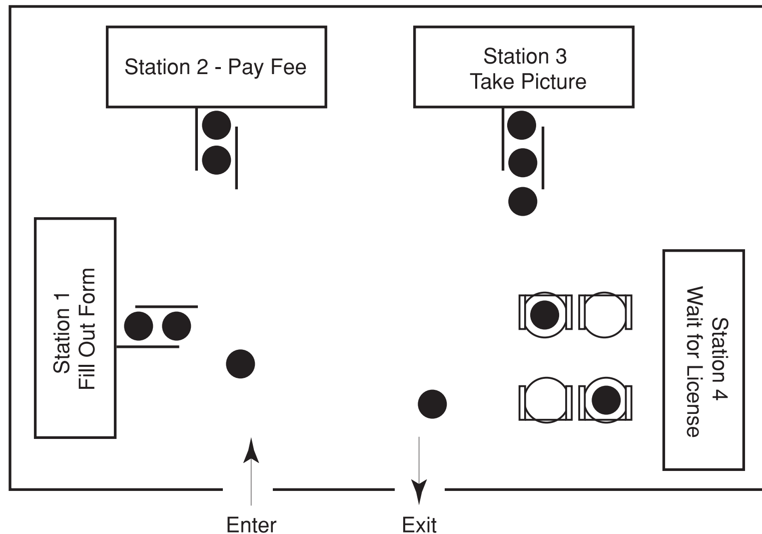


Figure 10-11. *"Getting Your Driver's License" as a multi-threaded program.*

Threads and COM

How does the concept of threads affect COM objects? Your program, be it single or multi-threaded, can use any COM object registered in the operating system. But if your multi-threaded object starts sending multiple threads through a COM object that doesn't protect its resources from multi-threaded access, trouble will surely result. Imagine if social mores broke down at the DMV and everyone tried to get their picture taken at the same time. Most likely, no one would ever get their picture taken successfully, and the whole process would grind to a halt. Similar things can happen inside a component in a situation like this.

What is needed, then, is a method for an object to inform the operating system and clients as to how it feels about threads. COM does this through the concept called *threading models*. There are four types of threading models, two of which will be discussed here.[3] Once instantiated on a thread, apartment-threaded objects may only be accessed by that same thread. This limitation to one thread is known as "thread affinity." Because they are limited to one thread, their resources need not be protected from multiple-thread access. Conversely, free-threaded objects may be accessed directly by any thread in the process. Because multiple threads may execute within the object simultaneously, all the resources in the object must be protected from multiple-thread access. Because of the lack of thread affinity, a free-threaded object is obviously much more flexible but, unfortunately, much more difficult to write. Visual Basic is unable to create free-threaded objects.

When an object is registered with the operating system, it tells the operating system what type of threading model it prefers. Depending on the threading model supported by the object, COM will load the object into something called an *apartment*. An apartment is kind of a "virtual area" that partitions how threads are allowed to access objects. An apartment model object is loaded into a Single-Threaded Apartment (STA). Only one thread is allowed to enter an STA and all the objects housed within it. Because any thread can instantiate an apartment-threaded object requiring an STA, a process can create as many STAs as it has threads. A free-threaded object is loaded into a Multi-Thread Apartment (MTA). MTAs and the objects within them can be accessed by any thread in the application. Because any thread can access an MTA, a process only needs to, in fact is only allowed to, create one MTA to service the whole process.

It is possible for threads to access other threads' STAs, but this access cannot be made directly. Instead, the requests made of objects in other apartments are marshalled across the apartment boundaries similar to the marshalling between processes discussed earlier. The same performance issues described in that section make cross apartment access a very unattractive situation for Web applications.

Threads, object scope, and IIS

If you've made it this far, congratulations! Now comes the payoff pitch: what all this means to your COM objects and IIS. Okay, read carefully and hang on; Internet Information Server maintains a pool of threads to service http requests. Each of these threads may access both its own STA and the IIS MTA directly. Nothing new so far; this is the same for all applications. IIS guarantees that a single thread will service an entire request. Because an object in Request scope can only be accessed by the thread servicing that request, that may reside either in the IIS MTA or in the thread's STA, meaning that the object may safely and efficiently be apartment threaded. An object in Session or Application scope can be accessed by any thread servicing a request for that Session or Application. Because IIS does not guarantee that all Session or Application requests will be serviced by the same thread, any object stored at Session or Application scope can be accessed by multiple threads. Further, because the only objects accessible by multiple threads directly are free-threaded objects in the MTA, all objects stored at Session or Application scope must be free threaded.

What does this mean from a practical standpoint? What kind of objects should you use for what functionality? Here's a quick set of rules to help you decide what type of functionality should be implemented where:

- Write free-threaded objects to store in the Application variable to hold state variables that are shared by all clients of the application. This

might include values like how many users are currently connected or have used the system in the past.

- Write free-threaded objects to store in the Session variable. These objects hold state variables that pertain to the user's current session. This might include values like how many items a user has ordered to this point.
- Write apartment-threaded objects to be instantiated in Request scope. Objects in Request scope are objects that are instantiated and released within the lifetime of one request. These objects might include functionality like data retrieval and mathematical calculations that have no need to keep a state longer than the duration of one request.

SUMMARY

In this chapter, you learned important rules to follow for implementing COM objects to run on your server. Here's a consolidated list of these rules. Keep these in mind as you read the rest of this book and develop your Web applications, and you will be a happier (or at least a better rested) developer.

1. Server-side COM objects may have no user interface.
2. Report user errors to the user and system errors to the system.
3. If at all possible, server-side objects should be in process.
4. Server-side objects must be as small as possible.
5. Handle errors according to COM conventions.
6. An object might not have the privileges you expect.
7. Learn the relationship between object scope and threads of execution.

---

[1] If your COM is a little rusty, check out Chapter 10, "COM Primer."

[2] According to the rules of COM, all methods return a type called an HRESULT. HRESULTs contain information on success or failure and the general nature of any errors that occurred. For more information on HRESULTs, see Chapter 10, "COM Primer."

[3] The other two threading models are single threaded and both threaded. Both-threaded objects support apartment threading and free threading and allow COM to choose the most appropriate model for the situation. Single-threaded objects are strictly limited to executing on the main thread of the application – such a loathsome context that we will speak of them no more!