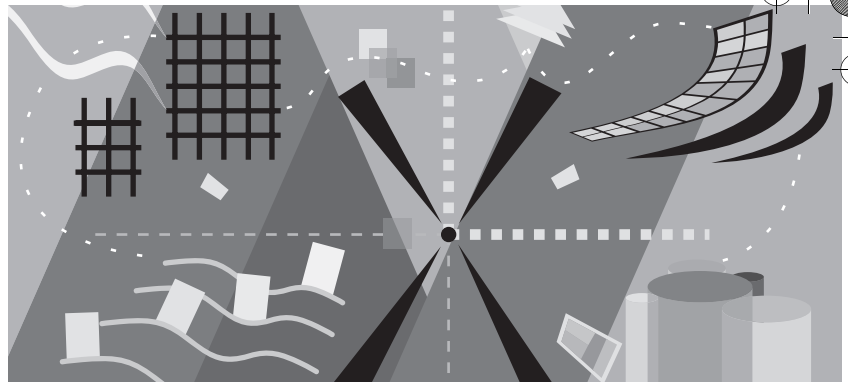


2



.NET Objects

*To match wi' Scotia's noblest speech yon orchestra sublime
 Whaurto-uplifted like the Just—the tail-rods mark the time.
 The Crank-throws give the double-bass; the feed-pump sobs an' heaves:
 An' now the main eccentrics start their quarrel on the sheaves.
 Her time, her own appointed time, the rocking link-head bides,
 Till-bear that note?—the rod's return whings glimmerin' through
 the guides.*

—Rudyard Kipling, writing about the vastly different types of components
 that any large application needs to work together
 harmoniously, “McAndrew’s Hymn,” 1894.

Problem Background

Good code is hard to write. It's never been easy, and the problems that developers need to solve to produce useful applications grow ever more complex in today's distributed, heterogeneous Internet world. I sometimes catch myself longing for the good old days, when software development meant writing an input processor that read characters directly from the keyboard and parsed them into recognizable tokens to be fed to a command processor. It doesn't work that way any more. Here are several of the difficult problems dogging the efforts of developers today.

First, we have the ongoing controversy over which programming language to use. While in theory any language can produce binary code that takes advantage of the entire operating system, it's all too common to hear something like, “Hey, you're using COBOL, so you can't have automatic

Good code is hard to write.

We need all system features to be available to programmers in any language.

14 Introducing Microsoft .NET, Third Edition

memory management. Get yourself a real language, kid,” or “Visual Basic doesn’t do uncouth things like threads” [nose in air]. We’d like the choice of language to be dictated by how well it matches the problem domain, not by how well it matches the system features. We don’t want any more second-class citizens. What’s probably going to be the downfall of the Java language is that you can only use its cool features from Java. I have no patience for anyone who insists that I embrace the One True Programming Language; instead, I believe that salvation ought to be available to believers of any development creed.

COM helped us develop applications by assembling purchased components; we didn’t have to write everything from scratch.

With the release of COM in 1993, Microsoft Windows developers found that they didn’t have to write all of their application’s code from scratch. COM allowed a client to call functions on a server at a binary level, without needing to know the server’s source code implementation. Using COM meant that we could buy components—say, a calendar control—from third-party vendors and wire them into our apps by writing a relatively thin layer of “glue” code to express our business logic. We got faster application development and better functionality than we could have written ourselves, and the third-parties got a much higher unit volume over which to amortize their development efforts. Microsoft also used COM to provide access to operating system functionality, such as queuing and transactions, again making apps faster and easier to write. It was a good idea, and the software gods smiled. For a while.

COM only went so far. We need to abstract away the differences in implementations.

As with most software architectures, COM helped to a certain point, but its internal structure has now become an obstacle rather than a help. COM has two main problems: First, it requires a substantial infrastructure from each application; for example, class factories and interface marshalers. Every development environment has to supply its own implementation of these mechanisms, so they’re all slightly different and not as compatible as we’d like. Second, COM operates by keeping client and server at arm’s length. They deal with each other through external interfaces, not through sharing their internal implementations. You might say that a COM client and server only make love by telephone. Unfortunately, everyone’s implementation of a COM interface differs in sneaky and hard-to-reconcile ways. For example, strings are implemented differently in C++ than they are in Microsoft Visual Basic, and both are implemented differently than strings in Java. Passing a string from a COM server written in Visual Basic to a COM client written in C++ requires work on someone’s part to iron out the differences, usually the C++ application because Visual Basic’s implementation isn’t negotiable. Programmers spend an inordinate amount of time ironing out these differences. That wastes valuable programmer time (and annoys programmers, making them change jobs to do something more fun), and you never know when you have it right, when any COM client regardless of implementation can use your

server. We need to iron out differences in implementation, allowing our apps to interoperate on a more intimate basis.

The Web is nothing if not heterogeneous. That's the dominant feature that any successful software architecture has to deal with. Much as Microsoft would like to see Windows PCs everywhere, they're starting to realize that it isn't going to happen. We'd like to be able to write software once and run it on a variety of platforms. That's what Java promised but hasn't quite delivered. (Spare me the righteous e-mails disagreeing with that statement; this is MY book.) Even if we can't make that approach work completely today, we'd like our software architecture to allow platform interoperability to evolve in the future.

We'd like our code to be able to run on a variety of platforms.

One of the major causes of program failure today, particularly in applications that run for a long time, is memory leaks. A programmer allocates a block of memory from the operating system, intending to free it later, but forgets and allocates another block. The first block of memory is said to be "leaked away," as it can't be recovered for later use. If your app runs long enough, these leaks accumulate and the app runs out of memory. That's not a big deal in programs like Notepad that a user runs for a few minutes and then shuts down, but it's fatal in apps like Web servers that are supposed to run continuously. You'd think we could remember to free all of our memory allocations, but they often get lost in complex program logic. Like an automatic seat belt that passengers couldn't forget to buckle, we'd like a mechanism that would prevent memory leaks in some way that we couldn't forget to use.

We need automatic memory management to prevent leaks.

When you ship a product, it's never perfect. (I know, yours are, but you'll have to agree that no one else's are, right? Besides, with no updates, how would you get more money from your existing customers?) So some time after you ship the first version, you ship an updated version of the product with new features and bug fixes for the old ones. Now the fun starts. No matter how hard you try to make your new release backward compatible with all of its old clients, this is very hard to do and essentially impossible to prove that you have done it. We'd really like some standardized mechanism whereby servers can publish the version level they contain. We'd like this mechanism to enable clients to read the version level of available servers and pick one with which they are compatible or identify exactly what they are missing if they can't.

We need help with managing different versions of the same software package.

Object-oriented programming, using such techniques as classes and inheritance, has permeated the software development world. That's about the only way you can manage programming efforts above a certain, not-very-high level of complexity. Unfortunately, every programming language provides a different combination of these features, naturally all incompatible,

We'd like object-oriented programming features to be available in and between all programming languages.

16 Introducing Microsoft .NET, Third Edition

which means that different languages can interoperate with each other only at a very low level of abstraction. For example, COM does not allow a Visual Basic programmer to use the convenient mechanism of inheritance to extend an object written in C++. Instead, COM requires cumbersome workarounds. We'd like object-oriented programming techniques to be available in and between all programming languages.

For safety, we want to be able to restrict the operations of pieces of code we don't fully trust.

The Web is fast becoming the main avenue by which users acquire software, which leads to major security problems. While current versions of Windows use digital certificates to identify the author of a piece of downloaded code, there is currently no way to ensure that a piece of code can't harm our systems, say, by scrambling files. We can choose to install or not install a downloaded component on our system, but there is no good way to restrict its activities once it's there. It's an all-or-nothing decision, and we really don't like that. We'd like some way of setting allowed and forbidden operations for various pieces of code and of having the operating system enforce those restrictions. For example, we might like to say that a piece of code we've just downloaded can read files but can't write them.

We need a better way of organizing operating system functions for better access.

The Windows operating system has grown almost unimaginably complex. From its humble beginnings as a Solitaire host with just a couple of hundred functions, it's mushroomed into a behemoth FreeCell host with over 5000 separate functions. You can't find the one you want simply by looking at an alphabetical list; it takes too long. Programmers manage complex projects by organizing their software into logical objects. We need a similar method of organizing the functionality of the operating system into logically related groups so that we have at least some chance of finding the function we want.

Our new object model needs to seamlessly interoperate with COM, both as client and as server.

Finally, I don't want to dump on COM too badly. It was revolutionary in its day, and we're going to have a lot of it with us for the foreseeable future. Just as the first color TV sets needed to also receive the black and white broadcasts that predominated at the time, so does whatever object model we start using need to seamlessly interoperate with COM, both as client and as server.

It should be obvious that this long list of requirements is far more than any application vendor can afford to develop on its own. We have reached the limit of our potentialities. To move into the Internet world, we need a higher power that can provide us with a world we can live in.

Solution Architecture

The solution is managed code, executing in the common language runtime.

The .NET Framework is Microsoft's operating system product that provides prefabricated solutions to these programming problems. The key to the framework is *managed code*. Managed code runs in an environment, called

the *common language runtime*, that provides a richer and more powerful set of services than the standard Win32 operating system, as shown in Figure 2-1. The common language runtime environment is the higher power that we have to turn our code over to in order to deal with the harsh, savage world that is modern Internet programming.

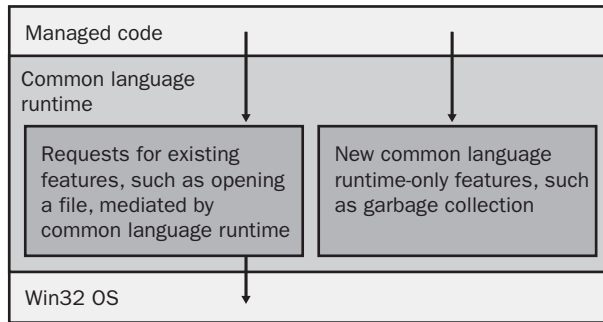


Figure 2-1 Managed execution in the common language runtime.

But with that architecture, how can the common language runtime work with any language? Not to sound Clintonesque, but that depends on what your definition of “language” is. Every common language runtime-compliant development tool compiles its own source code into a standard *Microsoft Intermediate Language* (MSIL, or IL for short), as shown in Figure 2-2. Because all development tools produce the same IL, regardless of the language in which their source code is written, differences in implementation are gone by the time they reach the common language runtime. No matter how it’s presented in the source code programming language itself, every program’s internal implementation of a string is the same as every other program’s because they all use the *System.String* object within the common language runtime. The same holds true for arrays and classes and everything else.

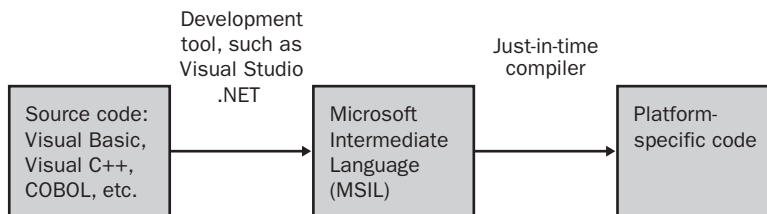


Figure 2-2 Different source code programming languages are compiled into MSIL.

18 Introducing Microsoft .NET, Third Edition

All common language runtime-compliant source code languages compile to the same intermediate language.

The IL is compiled just-in-time to run on the target machine.

The .NET Framework provides automatic memory management via garbage collection.

The .NET Framework supports explicit standardized version management.

The .NET Framework extends rich object-oriented programming features to all languages.

Any company that wants to can write a common language runtime-compliant language. Microsoft Visual Studio .NET provides common language runtime-compliant versions of Visual Basic, C# (pronounced C sharp), JScript, and C++. Visual Studio 2003 has added support for J#, which compiles Java language source code into .NET IL. Third parties are producing many others, including APL, COBOL, and Perl.

The IL code produced by the development tool can't run directly on any computer. A second step is required, called *just-in-time* (JIT) compilation, as shown in Figure 2-2. A tool called a *just-in-time compiler*, or JITter¹, reads the IL and produces actual machine code that runs on that platform. This provides .NET with a certain amount of platform independence, as each platform can have its own JITter. Microsoft isn't making a huge deal about just-in-time compiling, as Sun did about Java 5 years or so ago, because this feature is still in its infancy. No common language runtime implementations for platforms other than Windows (98, NT4 SP6, Me, 2000, or XP for clients; 2000 or XP Professional for servers) have currently been announced, although I expect some will be over time. It's probably more a strategy for covering future versions of Windows, like the forthcoming 64-bit version and now Windows XP, than it is for covering different operating systems, like Linux.

The .NET Framework provides automatic memory management, using a mechanism called *garbage collection*. A program does not have to explicitly free memory that it has allocated. The common language runtime detects when the program is no longer using the memory and automatically recycles it. Wish I had a maid that did that with my laundry.

The .NET Framework finally supports versioning. Microsoft .NET provides a standardized way in which developers of servers can specify the version that resides in a particular EXE or DLL and a standardized mechanism that a client uses to specify which version it needs to run with. The operating system will enforce the version requests of clients, both providing a reasonable default set of versioning behavior and allowing a developer to override it and specify explicit versioning behavior.

Because each language compiles to the same IL, all languages that support the common language runtime have the potential to support the same set of features. While it is possible to write a common language runtime language that does not expose this or that underlying common language runtime feature to a programmer, I expect the brutal Darwinian jungle that is the modern software marketplace to kill off such an ill-conceived idea very quickly. The common language runtime provides a rich set of object-oriented programming features, such as inheritance and parameterized object construction. Don't worry if these sound like complicated concepts—they aren't hard

1. Defects in this piece of software are known, of course, as jitterbugs.

to use; they save you a lot of time and potential errors; and you'll grow to like both of them.

The .NET Framework organizes operating system functionality through the *System* namespace. All operating system objects, interfaces, and functions are now organized in a hierarchical manner, so it's much easier to find the things you want. It also keeps your object and function names from colliding with those of the operating system and those of other developers.

The .NET Framework organizes system functionality into a hierarchical namespace.

The .NET Framework supports code access security. An administrator can specify that a piece of code is allowed to perform this operation but not that one. For example, you can allow a piece of code to read files but not write them, and the common language runtime will enforce your specifications and block any attempt to go outside them. This means that you can apply different levels of trust to code from different sources, just as you apply different levels of trust to different people that you deal with. This capability lets you run code from the Web without worrying that it's going to trash your system.

The .NET Framework supports code security.

Finally, the .NET Framework provides seamless interoperability with COM, both as client and as server. The framework puts a wrapper object around a COM object that makes the object look like a native .NET object. This means .NET code doesn't know or greatly care which kind of object it's running with. On the flip side, .NET objects know how to register themselves with an abstraction layer so that they appear to COM clients to be COM servers.

The .NET Framework provides seamless interoperability with COM, both as client and as server.

Oh Yeah? What Does It Cost?

But what about Platt's Second Law? (The amount of crap in the universe is conserved; see Chapter 1.) If I have less crap to deal with—for example, if I no longer have to worry about freeing memory that I've allocated—whose head did that crap get dumped on because it didn't just disappear? In the case of .NET, it got dumped primarily on two sets of heads, namely Microsoft's and Intel's. (All of my readers who work for Sun just stood up and cheered. Both of them.) In Microsoft's case, the operating system itself got harder to write. An automatic garbage collection mechanism like the one in .NET is several orders of magnitude harder to write than a simple in-out heap manager of the type that Windows 2000 contains. Since Microsoft hopes to sell millions of copies of .NET, they can afford to hire lots of smart programmers and engineer the heck out of it. This division of labor—letting Microsoft develop infrastructure while you worry less—makes economic sense.

The operating system got harder to write, but you don't really care about that.

In the case of Intel, the new .NET Framework will keep them busy producing faster CPUs and more memory chips. Applications built on .NET will run slower in some operations than those that aren't, but they'll be easier to write and debug. Sophisticated garbage collection requires more computation than simple heap allocation, just as an automatic seatbelt requires more parts

The computational task also got harder, requiring more computing horsepower.

than a manual one. Plus, since garbage collection doesn't take place as often, your computer probably needs more memory so that your app still has enough while objects are hanging around waiting to be garbage collected. (Remember Grosch's Law? Go check the end of Chapter 1 if you don't.) But I don't think that the additional memory and CPU cycles a .NET program requires are being squandered, as they are on that stupid dancing paper clip in Microsoft Office. I think they're being wisely invested, saving you time and money by letting you write code faster and with fewer bugs because the operating system is doing more of the scut work for you. An application using a general disk operating system will never run as fast as one that programs absolute sector and track disk head movements. But you can't afford to do that; it takes too long, it costs too much, and you can't manage very much data. You'll spend all your time on the silly disk sectors and never get any paying work done. Once it becomes possible to abstract away these infrastructural problems, doing so becomes an economic necessity. If your own memory management was working well enough, you wouldn't be spending your debugging time tracking memory leaks.

Simplest Example

A .NET Framework object sample begins here.

As I'll do throughout this book, I've written the simplest example I could think of to demonstrate the operation of the .NET Framework. You can download this sample and all the other code examples in this book from <http://www.introducingmicrosoft.net>. For this sample I wrote a .NET object server, the .NET replacement for an ActiveX DLL in Visual Basic 6, and its accompanying client. The server provides a single object exposing a single method, called *GetTime*, that provides the current system time in the form of a string, either with or without the seconds digits. Even though I wrote the server in Visual Basic and the client in C#, I didn't have to use Visual Studio. In fact, I wrote both applications in Notepad and built them with the command line tools provided in the .NET Framework SDK. I do show examples of using Visual Studio .NET in other sections of this chapter. Note: You can download a copy of the .NET Framework SDK at <http://www.msdn.microsoft.com/net>.

Visual Basic .NET contains a number of critical language differences from Visual Basic 6.

You'll notice when we begin looking at the code that it seems, at least superficially, quite similar to the classic Visual Basic code you are already familiar with. However, Microsoft made a number of important changes to the .NET version of Visual Basic to enable it to use the .NET common language runtime classes and interoperate correctly with the other common language runtime languages. A full discussion of these changes is far beyond the scope of this book, but here are two examples. The text string displayed in a button is now stored in a property called *Text* (as for a *TextBox*) rather than in the

Caption property used in Visual Basic 6. This change will break your existing app's compilation but is trivial to fix once the compiler shows it to you. Other changes won't break your compilation, but they can change your program's behavior in subtle and far-reaching ways. For example, a Visual Basic 6 object is destroyed immediately when its reference count reaches zero, but a zero-reference Visual Basic .NET object won't actually be destroyed until a garbage collection occurs, some indeterminate amount of time later. (See the discussion later in this chapter.) Your app might be able to live with the new behavior, or it might require a redesign. These changes mean that you cannot simply compile your existing Visual Basic code in Visual Studio .NET and expect it to work correctly. It will take some effort to port; probably not an enormous amount, but more than the zero-level you were hoping for.

Note Visual Studio .NET contains an upgrade tool that runs automatically when you open a Visual Basic 6 project. It flags the changes that it detects and suggests fixes. The language has definitely gotten more powerful. If you want the cool Internet features of .NET, you'll probably think it's worth the effort to switch. Even if you're just writing single-user desktop form applications, you may still find the versioning support and the easier deployment and cleanup to be worth it.

Listing 2-1 shows the code listing for my sample object server. Looking at this code, we first see the *Imports* directive. This new feature of Visual Basic .NET tells the compiler to "import the namespaces." The term *namespace* is a fancy way to refer to the description of a set of prefabricated functionality provided by some class somewhere. It is conceptually identical to a reference in your Visual Basic 6 project. The names following *Imports* tell the engine which sets of functionality to include the references for. In this case, *Microsoft.VisualBasic* is the one containing the definition of the *Now* function that I use to fetch the time. If you use Visual Basic from within Visual Studio .NET, the *Microsoft.VisualBasic* namespace is imported automatically without needing an explicit *Imports* statement.

We next see the directive *Namespace TimeComponentNS*. This is the declaration of the namespace for the component we are writing, the name that clients will use when they want to access this component's functionality. I discuss namespaces later in this chapter. Again, if you are using Visual Studio .NET, this declaration is made automatically.

22 Introducing Microsoft .NET, Third Edition

```
' Import the external Visual Basic namespace, allowing me to
' access the Now function by its short name.

Imports Microsoft.VisualBasic

' Declare the namespace that clients will use to access
' the classes in this component.

Namespace TimeComponentNS

' Declare the class(es) that this DLL will provide to a client.
' This is the same as Visual Basic 6.

Public Class TimeComponent

' Declare the function(s) that this class will provide to a client.
' This, too, is the same as VB6.

Public Function GetTime(ByVal ShowSeconds As Boolean) As String

' The formatting of dates, and the returning of values of
' functions, changed somewhat in Visual Basic .NET.

If (ShowSeconds = True) Then
Return Now.ToLongTimeString
Else
Return Now.ToShortTimeString
End If

End Function

End Class

End Namespace
```

Listing 2-1 Visual Basic code listing of simplest object server.

This section describes the code of my .NET object server.

Next come the class and function declarations, identical to Visual Basic 6. Finally, I put in the internal logic of fetching the time, formatting it into a string and returning it to the client. These too have changed slightly. The property *Now* still fetches the date, but formatting it into a string is now done with a method of the new .NET class *DateTime* rather than a separate function. Also, a Visual Basic function specifies its return value using the new keyword *Return* instead of the syntax used in earlier versions.

I next compiled my code into a DLL, named `TimeComponent.dll`, using the command line tools that come with the .NET Framework SDK. Anyone who cares to can find the command line syntax in `Makecomponent.bat`, which you can download from the book's Web site. The result may look like a plain old DLL to you, but it's actually very different inside. The Visual Basic .NET compiler didn't convert the Visual Basic code to native code; that is, to specific instructions for the microprocessor chip inside your PC. Instead, the DLL contains my object server's logic expressed in MSIL (again, for Microsoft Intermediate Language; IL for short), the intermediate language that I introduced in the "Solution Architecture" section in this chapter. All common language runtime language compilers produce this IL rather than native processor instructions, which is how the runtime can run seamlessly with so many different languages. The DLL also contains *metadata* that describes the code to the common language runtime system. This metadata is in a runtime-required format that describes the contents of the DLL: what classes and methods it contains, what external objects it requires, what version of the code it represents, and so on. Think of it as a type library on steroids. The main difference is that a COM server could sometimes run without a type library, whereas a .NET object server can't even begin to think about running without its metadata. I discuss this metadata further in the section "Assemblies" later in the chapter.

Having written my server, I next need a client to test it. To demonstrate the fact that .NET works between different languages, I wrote this client in C#. Rather than convert from Visual Basic 6 to Visual Basic .NET, many of my customers are converting directly to C#. If you look at the code in Listing 2-2, you'll see that it's fairly easy to understand at this level of simplicity. In fact, given the enhancements to Visual Basic .NET to support the common language runtime's object-oriented features such as inheritance (described later in this chapter), I've heard programmers after a few beers describe C# as "VB with semicolons" or occasionally "Java without Sun." Either one of these can start a fistfight if you say it too loudly in the wrong bar in Redmond or Sunnyvale.

Our client example starts with importing namespaces, which in C# requires the directive *using*. Our sample client imports the *System* namespace (described in detail later in this chapter), which contains the description of the *Console.WriteLine* function, and also imports our time component's namespace. Additionally, we have to explicitly tell the compiler in which DLL it will find our component's namespace, which we do in the compiler batch file `Makeclient.bat` (not shown). Visual Studio provides an easy user interface for this.

Compiling the Visual Basic code produces a DLL containing intermediate language and metadata.

Visual Basic and C# resemble each other more than either community likes to admit.

24 Introducing Microsoft .NET, Third Edition

```
// Import the namespaces that this program uses, thereby allowing
// us to use the short names of the functions inside them.

using System ;
using TimeComponentNS ;

class MainApp
{

    // The static method "Main" is an application's entry point.

    public static void Main()
    {

        // Declare and create a new component of the class
        // provided by the VB server we wrote.

        TimeComponent tc = new TimeComponent ( ) ;

        // Call the server's GetTime method. Write its
        // resulting string to a console window.

        Console.Write (tc.GetTime (true)) ;

    }
}
```

Listing 2-2 C# code listing of simplest object client.

The C# client also compiles to intermediate language.

Execution of any C# program begins in a static (shared) method called *Main*. In that method, we can see that our client program uses the C# *new* operator to tell the runtime engine to find the DLL containing our *TimeComponent* class and create an instance of it. The next line calls the object's *GetTime* method and then uses the *System* namespace's *Console.Write* method to output the time string in a command line window. The C# compiler in this case produces an EXE file. Like the server DLL, this EXE does not contain native instructions, but instead contains intermediate language and metadata.

When I run the C# client executable, the system loader notes that the executable is in the form of managed code and loads it into the runtime engine. The engine notes that the EXE contains IL, so it invokes the just-in-time compiler, or JITer. As I discussed earlier, the JITer is a system tool that converts IL into native code for whichever processor and operating system it runs on. Each different architecture will have its own JITer tailored to that particular system, thereby allowing one set of IL code to run on multiple types of systems. The JITer produces native code, which the common language runtime engine will begin to execute. When the client invokes the *new* operator to create an object of the *TimeComponent* class, the common language runtime

engine will again invoke the JITer to compile the component DLL's IL just-in-time and then make the call and report the results. The output is shown in Figure 2-3.

```

C:\NET\book\Code\Chapter 2\Simplest>dir
03/23/2001 10:26a <DIR> .
02/05/2001 09:11a 47 makeclient.bat
02/05/2001 08:46a 31 makecomponent.bat
02/05/2001 09:42a 586 timeclient.cs
02/05/2001 09:36a 3,072 timeclient.exe
03/23/2001 10:26a 3,072 TimeComponent.dll
02/05/2001 09:39a 877 TimeComponent.vb
6 File(s) 7,685 bytes
2 Dir(s) 4,949,780,480 bytes free

C:\NET\book\Code\Chapter 2\Simplest>timeclient
C:\NET\book\Code\Chapter 2\Simplest>makeclient
C:\NET\book\Code\Chapter 2\Simplest>csc /reference:timecomponent.dll timeclient.cs
Microsoft (R) Visual C# Compiler Version 7.00.9148 [CLR version v1.0.2615]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\NET\book\Code\Chapter 2\Simplest>timeclient
10:26:45 AM
C:\NET\book\Code\Chapter 2\Simplest>ildasm timecomponent.dll
C:\NET\book\Code\Chapter 2\Simplest>

```

Figure 2-3 Console output of sample TimeClient program.

This run-time compilation model works well for some classes of applications, such as code downloaded from the Internet for a page you just surfed to, but not for others, say, Visual Studio, which you use all day, every day, and update once or twice a year. Therefore, an application can specify that JIT compilation is to be performed once, when the application is installed on a machine, and the native code stored on the system as it is for non-.NET applications. You do this via the command-line utility program Ngen.exe, the native image generator, not shown in this example.

When the client used the *new* operator to create the object, how did the loader know where to find the server DLL? In this case, the loader simply looked in the same directory as the client application. This is known as a *private assembly*, the simplest type of deployment model in .NET. A private assembly can't be referenced from outside its own directory. It supports no version checking or any security checking. It requires no registry entries, as a COM server would. To uninstall a private assembly, all you have to do is delete the files, without performing any other cleanup. Obviously, this simple case, while effective in situations like this, isn't useful in every situation—for example, when you want to share the same server code among multiple clients. I discuss these more complex scenarios in the section on assemblies later in this chapter.

The IL is compiled just-in-time when the client and its component are run.

The loader finds the DLL requested by the client by looking in the client application's directory.

More on .NET Namespaces

I remember programming Windows version 2.0, scanning the alphabetical list of operating system functions (on paper, that's how long ago this was) until I found the one whose name seemed to promise that it would do what I

Selecting items from a short alphabetical list is easy. It's much harder when the list gets longer.

26 Introducing Microsoft .NET, Third Edition

wanted. I'd try it, and sometimes it would work and sometimes it wouldn't. If it didn't, I'd go back to scanning the list again. Listing the functions alphabetically worked reasonably well on Windows 2.0, which contained only a few hundred different functions. I could see at a glance (or two or three) everything that the operating system could do for me, which gave me a fighting chance at writing some decent, albeit limited, code.

Organizing operating system functions into one alphabetical list won't work with today's 32-bit Windows. It's enormous—over 5000 functions and growing. I can't scan through a list that long to find, for example, console output functions; they're scattered among far too many unrelated functions for me to pick them out. It's a problem for operating system designers, too. When they want to add a new function, they have to choose a name for it that is descriptive but that doesn't conflict with any of the other function names already implemented. Application programmers also need to make sure that their global function names don't conflict with operating system functions. The signal-to-noise ratio of this approach gets lower as the list of functions gets longer, and it's approaching absolute zero today. We say that the *namespace*, the set of names within which a particular name needs to be unique, has gotten too large.

The way to handle large lists is to break them down into smaller sublists that you can more easily digest. The classic example of this is the Start menu in Windows. If every application on your entire computer were listed on one gigantic menu, you'd never be able to find the one that you wanted. Instead, the Start menu provides a relatively short (10 or so) list of groups, easy to scan and pick from. Each group contains a list of logical subgroups, nested as deeply as you feel is cost-effective, eventually terminating in a short list of actual applications. By the time you pick the application you want, you've looked at maybe 50 different choices, rarely more than a dozen or so at a time. Think how much easier this is compared to selecting the one application you want out of the thousand or so installed on most computers.

The .NET Framework provides a better way of organizing operating system functions and objects. This same mechanism keeps the names of functions and objects that you write from interfering with the names of objects and functions written by other developers. It uses the concept of a namespace, which is a logical subdivision of software functionality within which all names must be unique. It's not a new concept; object-oriented languages have used it for decades. But its use in .NET is the first time I know of that an entire operating system's functionality has been organized in this way.

All .NET common language runtime objects and functions are part of a namespace called *System*. When you look them up in the documentation, you'll find that all names begin with the characters "System." We say that all

The best way to handle large lists is to break them down into smaller logical groups.

.NET provides the concept of a namespace, a logical division within which a name needs to be unique.

All .NET common language runtime objects and functions live within the *System* namespace.

the objects and functions whose names begin this way “belong to the *System* namespace.” The *System* namespace is naturally quite large, as it contains the names of all functional elements of a rich operating system. It is therefore subdivided into a number of subordinate namespaces—for example, *System.Console*, which contains all the functions dealing with input and output on a console window. Some of these subnamespaces contain their own sub-subnamespaces, and so on down the line until the developers got tired of it. The *fully qualified name* of a function, sometimes called the *qualified name* or *q-name*, is the name of the function preceded by its full namespace location, also known as a *qualifier*. For example, *System.Console.Write* is the fully qualified name of the system function that writes output to a console window. You can call a function by means of its fully qualified name from anywhere in your code.

The *System* namespace is very large. Consequently, it is implemented in several separate DLLs. Just because a function or an object is part of the *System* namespace does not necessarily mean that your editor, compiler, and linker will automatically be able to find it. You generally have to tell your development tools which of the *System* namespace DLLs you want to include when building a project. For example, when I build .NET components in Visual Studio, I often like them to pop up message boxes during debugging. The *MessageBox* object is part of the *System.Windows.Forms* namespace, which is implemented in *System.Windows.Forms.dll*. Visual Studio does not automatically include this DLL in its reference list when I create a component library project, probably because its authors figured that components would work behind the scenes and not interact with the user. That’s generally true in production. To gain access to the *MessageBox* object during debugging, I have to explicitly add that reference to my project.

This organization of functions into logical groups is very handy for finding the one you want with a minimum of fuss. The only drawback is that fully qualified names can get very long. For example, the function *System.Runtime.InteropServices.Marshal.ReleaseComObject* is used for releasing a specified COM object immediately without performing a full garbage collection. (See the section “.NET Memory Management” for an explanation of the latter.) Most .NET applications won’t use this function at all, but the ones that do will probably use it in many places. Typing this whole thing in every time you call it could get very tedious very quickly. You can see that it barely fits on one line of this book. My wife does not address me as “David Samuel Platt, son of Benjamin, son of Joseph” unless she is exceptionally angry, a state she is incapable of occupying for very long. Therefore, just as people address their most intimate relations by their first names, the common language runtime allows you to *import a namespace*, as shown earlier in the programming examples

The *System* namespace is implemented in several separate DLLs. You have to be sure that your development tools know to include all the ones you need.

Importing a namespace allows you to use short names when calling a function within that namespace.

in Listings 2-1 and 2-2. When you import a namespace, you're telling your compiler that you use the functions in that namespace so often that you want to be on a first-name basis with them. Importing a namespace is done in Visual Basic by using the keyword *Imports*, and in C# via the keyword *using*. For example, in Listing 2-2, I imported the namespace *System*, which allowed me to write to the console by calling *Console.Write*. If I hadn't imported the *System* namespace, I would have had to use the fully qualified name *System.Console.Write*. Choosing which namespaces to import is entirely a matter of your own convenience and has no effect on the final product (IL always uses full names) other than allowing you to organize your own thought processes so as to produce your best output. Since the whole point of using separate namespaces is to separate functions with the same name to prevent conflicts, I'd suggest that you not import them all at the same time. Instead, I strongly urge you to pick a consistent set of rules for choosing which namespaces to import and follow it throughout your entire project.

Your own code will also live in a namespace, with a name you define.

When you write a .NET object server, you specify the name of the namespace in which your code lives. This is done via the *Namespace* directive, as shown in Listing 2-1. The namespace will often be the same as the name of the file in which the code lives, but it doesn't have to be. You can put more than one namespace in the same file, or you can spread one namespace among multiple files. Visual Studio .NET or other development environments will often automatically assign a namespace to your project. But how do you know that your namespace won't conflict with the namespace chosen by another vendor for a different component? That's what assemblies are for, which is our next topic of conversation.

Tips from the Trenches

A number of my customers report that they're happy using the following rule: If you refer to a namespace three or more times within a source code file, import that namespace. They find it easy to remember and follow, and they like the familiarity it engenders. On the other hand, when I'm writing sample code, I often don't import any namespaces other than *System*, using fully qualified names at all times to emphasize exactly which part of the .NET system a particular object belongs to.

Assemblies

The .NET Framework makes extensive use of *assemblies* for .NET code, resources, and metadata. All code that the .NET common language runtime executes must reside in an assembly. In addition, all security, namespace resolution, and versioning features work on a per-assembly basis. Since assemblies are used so often and for so many different things, I need to discuss assemblies in some detail.

.NET makes extensive use of a new packaging unit called an assembly.

Concept of an Assembly

An assembly is a logical collection of one or more EXE or DLL files containing an application's code and resources. An assembly also contains a *manifest*, which is a metadata description of the code and resources “inside” the assembly. (I'll explain those quotes in a second.) An assembly can be, and often is, a single file, either an EXE or a DLL, as shown in Figure 2-4.

When we built the simple example of a time server earlier in this chapter, the DLL that our compiler produced was actually a single-file assembly, and the EXE client application that we built in that example was another one. When you use tools such as Visual Studio .NET, each project will most likely correspond to a single assembly.

Our simple example produced two single-file assemblies.

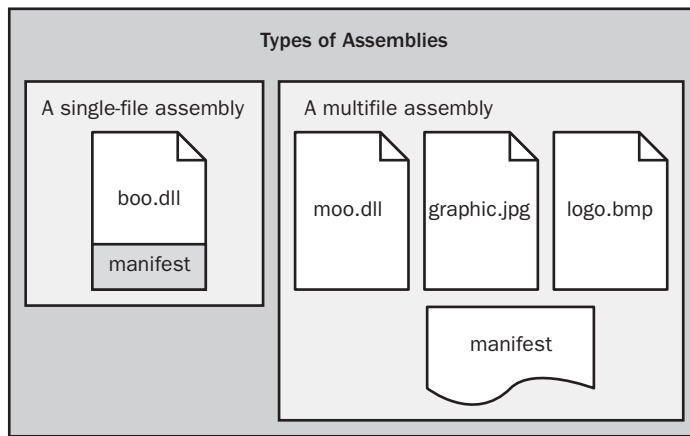


Figure 2-4 Single-file and multifile assemblies.

Although an assembly often resides in a single file, it also can be, and often is, a logical, not a physical, collection of more than one file residing in the same directory, also shown in Figure 2-4. The manifest specifying the files

An assembly can also be a logical collection of more than one file.

30 Introducing Microsoft .NET, Third Edition

that make up the assembly can reside in one of the code-containing EXEs or DLLs of the assembly, or it can live in a separate EXE or DLL that contains nothing but the manifest. When dealing with a multifile assembly, you *must* remember that the files are not tied together by the file system in any way. It is entirely up to you to ensure that the files called out in the manifest are actually present when the loader comes looking for them. The only thing that makes them part of the assembly is that they are mentioned in the manifest. In this case, the term *assembly*, with its connotation of metal parts bolted together, is not the best term. Perhaps “roster” might be a better one. That’s why I put quotes around the term “inside” the assembly a few paragraphs ago. You add and remove files from a multifile assembly using the command line SDK utility program AL.exe, the Microsoft Assembly Linker.

You can view an assembly’s manifest with ILDASM.exe

You can view the manifest of an assembly using the IL Disassembler (ILDASM.exe). Figure 2-5 shows the manifest of our time component. You can see that it lists the external assemblies on which this assembly depends. In this case, we depend on mscorlib.dll, the main .NET common language runtime DLL, and on an assembly called Microsoft.VisualBasic, which contains Visual Basic’s internal functions such as *Now*. It also lists the assembly names that we provide to the world, in this case, TimeComponent.

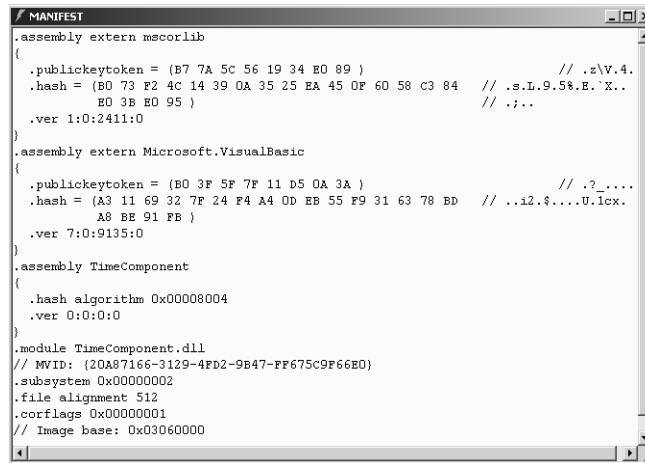


Figure 2-5 Assembly manifest of our sample time component.

In addition to the code objects exposed by and required by the assembly, the manifest also contains information that describes the assembly itself. For example, it contains the assembly’s version information, expressed in a standardized format described later in this section. It can also describe the culture (fancy name for human language and sublanguage, say, Australian English) for

which the assembly is written. In the case of a shared assembly, of which more anon, the manifest also contains a public cryptographic key, which is used to ensure that the assembly can be distinguished from all other assemblies regardless of its filename. You can even add your own custom attributes to the manifest, which the common language runtime will ignore, but which your own applications can read and use. You set manifest attributes with the Assembly Linker mentioned previously or with Visual Studio.

Assemblies and Deployment

The central question in dividing your code among assemblies is whether the code inside the assembly is intended solely for your own application's use or will be shared with any other application that wants it. Microsoft .NET supports both options, but it requires more footwork in the latter case. In the case of code that you write for your own applications, say, the calculation engine for a complex financial instrument, you'd probably want to make the assembly private. On the other hand, a general utility object that could reasonably be used by many applications—a file compression engine, for example—might be more widely used if you make it shared.

Suppose you want your assemblies to be private. The .NET model couldn't be simpler. In fact, that's exactly what I did in the simplest example shown previously. You just build a simple DLL assembly, and copy it to the directory of the client assembly that uses it or to a subdirectory of that client. You don't have to make any entries in the system registry or Active Directory as you had to do when using COM components. None of the code will change unless you change it, so you will never encounter the all-too-familiar situation in which a shared DLL changes versions up or down and your app breaks for no apparent reason.

The obvious problem with this approach is the proliferation of assemblies, which was the problem DLLs were originally created to solve back in Windows 1.0. If every application that uses, say, a text box, needs its own copy of the DLL containing it, you'll have assemblies breeding like bacteria all over your computer. Jeffrey Richter argued (in *MSDN Magazine*, March 2001) that this isn't a problem. With 40 gigabyte hard drives selling for under \$200 (then; today for \$200 you can get 200 GB), everyone can afford all the disk space they need, so most assemblies should be private; that way your application will never break from someone else messing with shared code. That's like an emergency room doctor saying that the world would be a far better place if people didn't drink to excess or take illegal drugs. They're both absolutely right, but neither's vision is going to happen any time soon in the real world. Richter's idea is practical for developers, who usually get big, fast PCs,

You need to think carefully about whether your assemblies should be private or public.

Assemblies can be private to an application, which simplifies your life in certain cases.

However, sometimes you want the code in assemblies to be shared.

32 Introducing Microsoft .NET, Third Edition

but a customer with a large installed base of two-year-old PCs that it can't junk or justify upgrading at that point in its budget cycle isn't going to buy that argument *or* bigger disks. Fairly soon in your development process, you will need to share an assembly among several applications, and you want the .NET Framework to help you do that.

Shared assemblies live in the global assembly cache, administered by a number of tools.

The .NET Framework allows you to share assemblies by placing them in the *global assembly cache* (GAC, pronounced like the cartoon exclamation). This is a directory on your machine, currently \winnt\assembly or \windows\assembly, in which all shared assemblies are required to live. You can place assemblies into the cache, view their properties, and remove them from the cache using a .NET Framework SDK command line utility called GACUTIL.exe, which works well when run from scripts and batch files. Most human users will prefer to use the Assembly Cache Viewer, which is a shell extension that installs with the .NET Framework SDK. It automatically snaps into Windows Explorer and provides you with the view of the GAC shown in Figure 2-6.

Shared assemblies use public key cryptography to ensure that their names are unique.

Whenever you share any type of computer file, you run up against the problem of name collisions. Because all .NET shared assemblies have to go in the GAC so that they can be managed, we need some way of definitively providing unique names for all the code files that live there, even if their original file names were the same. This is done with a *strong name*, otherwise known as a *shared name*. A strong name uses public key cryptography to transparently produce a name that is guaranteed to be unique among all assemblies in the system. The manifest of a shared assembly contains the public key of a public/private key pair. The combination of the file's name, version, and an excerpt from this public key is the strong name.

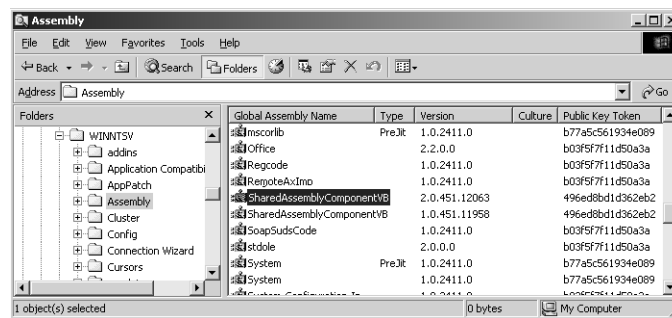


Figure 2-6 Global assembly cache viewer.

Suppose we want to write a shared assembly that lives in the GAC. I've switched to Visual Studio .NET for this example, both to demonstrate it and because I find it easier to operate than the command line tools. I've written a different .NET component that does the same thing as our simplest time example, except that it adds the version number to its returned time string. Once I build the component, I need to generate and assign a strong name for it, also known as *signing* the component. Visual Studio .NET can be configured to do this automatically if you provide a file containing the public/private key pair. You generate this file with the SDK command line utility program SN.exe. You tell Visual Studio about the key file by specifying the filename in the AssemblyInfo.vb file in the project, as shown in Listing 2-3. When I build the component, Visual Studio .NET signs it automatically. I then manually put it in the GAC by using Windows Explorer.

This paragraph contains instructions for generating a shared assembly.

```
<Assembly: AssemblyKeyFileAttribute("../..\\mykeys.snk")>
```

Listing 2-3 AssemblyInfo.vb file entry specifying key pair for generating strong name.

I've also provided a client that uses the shared assembly. I tell Visual Studio to generate a reference to the server DLL by right-clicking on the References folder in Solution Explorer, selecting Add Reference to open the Add Reference dialog box (shown in Figure 2-7), and then clicking Browse and surfing over to the shared assembly file that resides in a standard directory. Visual Studio generates a reference accessing that assembly.

This paragraph contains instructions for writing a client that uses an object from the GAC.

Visual Studio cannot currently (version 2003) add a reference to an assembly in the GAC, although this feature has been proposed for a future release. This happened because in the first version of Visual Studio .NET, the GAC's design hadn't yet stabilized by the time the developers needed to design their reference mechanism. (Why they haven't fixed this in Visual Studio .NET 2003 isn't clear.) Therefore, unless they're building client and server together as part of the same project, developers must install two copies of their components, one in a standard directory to compile against and another in the GAC for their clients to run against. Users will require only the latter. When you add a reference to an assembly marked with a strong name, Visual Studio automatically sets the *CopyLocal* property of the newly-added reference to False, thereby telling Visual Studio that you don't want it to make a local copy. It figures that, since the assembly has a strong name and is therefore able to go into the GAC, that you probably want to run with the GAC copy.

34 Introducing Microsoft .NET, Third Edition

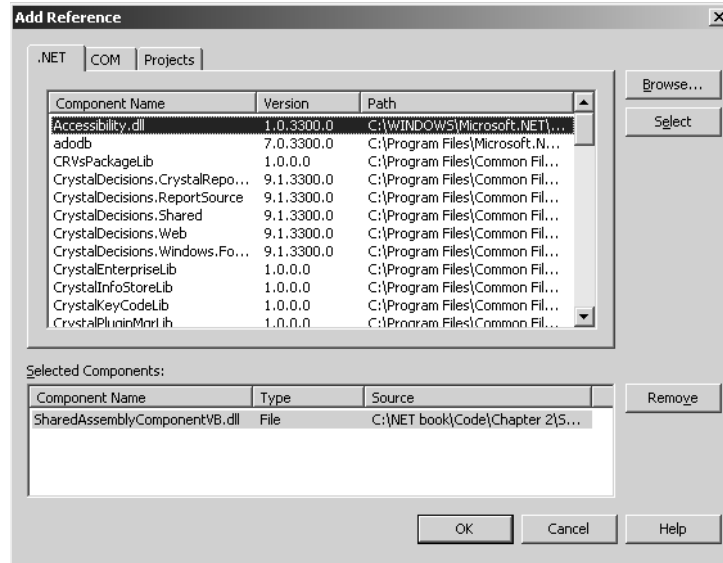


Figure 2-7 Adding a reference to a shared component.

The public/private key algorithm also provides a check on the integrity of the assembly's files.

As an added benefit of the public key cryptography scheme used for signing shared assemblies, we also gain a check on the integrity of the assembly file. The assembly generator performs a hashing operation on the contents of the files contained in the manifest. It then encrypts the result of this hash using our private key and stores the encrypted result in the manifest. When the loader fetches an assembly from the GAC, it performs the same hashing algorithm on the assembly's file or files, decrypts the manifest's stored hash using the public key, and compares the two. If they match, the loader knows that the assembly's files haven't been tampered with. This doesn't get you any real identity checking because you can't be sure whose public key it really is, but it does guarantee that the assembly hasn't been tampered with since it was signed.

Assemblies and Versioning

Versioning of code is an enormous, painful, unsexy problem.

Dealing with changes to published code has historically been an enormous problem, often known as DLL Hell. Replacing with a newer version a DLL used by an existing client bit you two ways, coming and going. First, the new code sometimes broke existing applications that depended on the original version. As hard as you try to make new code backward compatible with the old, you can never know or test everything that anyone was ever doing with it. It's especially annoying when you update a new DLL and don't run the now-broken old client until a month later, when it's very difficult to remember what you might have done that broke it. Second, updates come undone

when installing an application copies an older DLL over a newer one that's already on your computer, thereby breaking an existing client that depended on the newer behavior. It happens all the time, when an installation script says, "Target file xxx exists and is newer than the source. Copy anyway?" and 90 percent of the time the user picks Yes. This one's especially maddening because someone else's application caused the problem, but your app's the one that won't work, your tech support line is the one that receives expensive calls and bomb threats, and you better hope you haven't sold any copies of the program to the Postal Service. Problems with versions cost an enormous amount of money in lost productivity and debugging time. Also, they keep people from buying upgrades or even trying them because they're afraid the upgrade will kill something else, and they're often right.

Windows has so far ignored this versioning problem, forcing developers to deal with it piecemeal. There has never been, until .NET, any standardized way for a developer to specify desired versioning behavior and have the operating system enforce it. In .NET, Microsoft seems to have realized that this is a universal problem that can be solved only at an operating system level and has provided a system for managing different versions of code.

Every assembly contains version information in its manifest. This information consists of a *compatibility version*, which is a set of four numbers used by the common language runtime loader to enforce the versioning behavior requested by a client. The compatibility version number consists of a major and minor version number, a build number, and a revision number. The development tools that produce an assembly put the version information into the manifest. Visual Studio .NET produces version numbers for its assemblies from values that you set in your project's AssemblyInfo.vb (or .cs) file, as shown in Listing 2-4. Command line tools require complex switches to specify an assembly's version. You can see the version number in the IL Disassembler at the bottom of Figure 2-8. You can also see it when you install the assembly in the GAC, as shown previously in Figure 2-6.

.NET finally incorporates some functionality for versioning.

Each assembly contains information telling the runtime what version number it represents.

```
' Version information for an assembly consists of the following
' four values:
'
'     Major Version
'     Minor Version
'     Revision
'     Build Number
'
' You can specify all the values or you can default the Build and
' Revision Numbers by using the '*' as shown below:

<Assembly: AssemblyVersion("2.0.*")>
```

Listing 2-4 AssemblyInfo.vb file showing version of component assembly.

36 Introducing Microsoft .NET, Third Edition

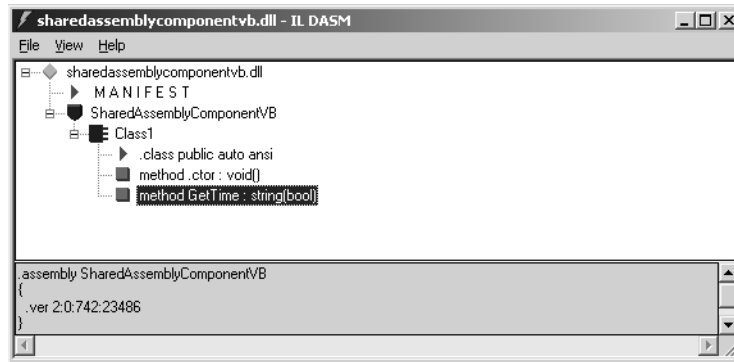


Figure 2-8 ILDASM showing version of a server component.

The manifest can also contain an *informational version*, which is a human readable string like “Microsoft .NET 1.1 April 2003.” The informational version is intended for display to human viewers and is ignored by the common language runtime.

When you build a client assembly, you’ve seen that it contains the name of the external assemblies on which it depends. It also contains the version number of these external assemblies, as you can see in Figure 2-9.

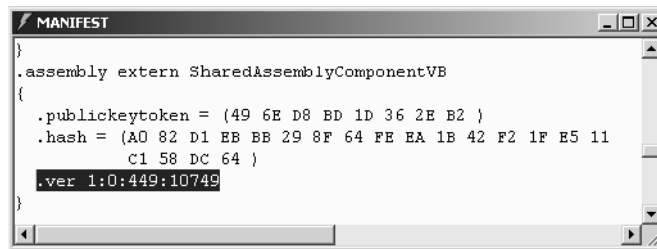


Figure 2-9 ILDASM showing required version in a client.

Every client assembly contains information about the versions it was built against.

By default, a client requires the exact version of the server against which it was built.

You can override default versioning behavior by using configuration files.

When the client runs, the common language runtime looks to find the version that the client needs. The default versioning behavior requires the exact version against which the client was built; otherwise the load will fail. Since the GAC can contain different versions of the same assembly, as shown in Figure 2-6, you don’t have the problem of a new version breaking old clients, or an older version mistakenly replacing a new one. You can keep all the versions that you need in the GAC, and each client assembly will request and receive the one that it has been written and tested against.

Occasionally this exact-match versioning behavior isn’t what you want. You might discover a fatal defect, perhaps a security hole, in the original version of the server DLL, and need to direct the older clients to a new one

immediately. Or maybe you find such a bug in the new server and have to roll the new clients back to use the old one. Rather than have to recompile all of your clients against the replacement version, as would be the case with a classic DLL, you can override the system's default behavior through the use of configuration files.

The most common way to do this is with a *publisher policy*, which changes the versioning behavior for all clients of a GAC assembly. You set a publisher policy by making entries in the master configuration file `machine.config`, which holds the .NET administrative settings for your entire machine. `Machine.config` is an XML-based file, and you might be tempted to go at it with Notepad or your favorite XML editor. I strongly urge you to resist this temptation; wipe out one angle bracket by accident or get the capitalization wrong on just one name and your entire .NET installation may become unusable (shades of the registry, except no one used Notepad on that, at least not for long). Instead, use the .NET Framework Configuration utility `mscorcfg.msc`, shown in Figure 2-10, which comes with the .NET SDK. This utility allows you to view the GAC, similar to the Windows Explorer add-in I showed in Figure 2-6. In addition, it allows you to configure the behavior of assemblies in the GAC.

A publisher policy changes the versioning behavior of a GAC assembly for all its clients.

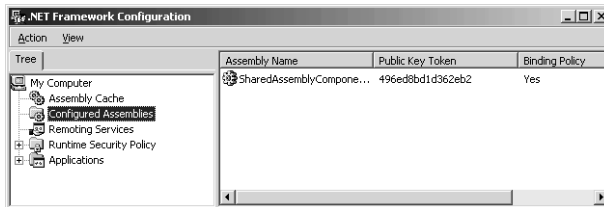


Figure 2-10 The .NET Framework Configuration utility.

You set a publisher policy by making your server assembly into a *configured assembly*, which is a GAC assembly for which a configuration file holds entries that change the assembly's behavior from default GAC assembly behavior. You do this by right-clicking the `Configured Assemblies` tree item, selecting `Add`, and either entering a specific assembly or selecting the assembly you want from the list you're offered. Once you have made the assembly into a configured assembly, you can then change its behavioral properties by right-clicking on the assembly in the right-hand pane and choosing `Properties` from the context menu. Figure 2-11 shows the resulting dialog box. You enter one or more binding policies, each of which consists of a set of one or more old versions that the assembly loader will map to exactly one new version. The configuration utility will write these entries into the configuration file in the proper format. When a client creates an object requesting one of the

You set a publisher policy using the .NET Framework Configuration utility `mscorcfg.msc`.

38 Introducing Microsoft .NET, Third Edition

specified older versions, the loader checks the configuration file, detects the publisher policy, and automatically makes the substitution. You can also enter a codebase for an assembly, which tells the loader from where to download a requested version if it isn't already present on the machine.

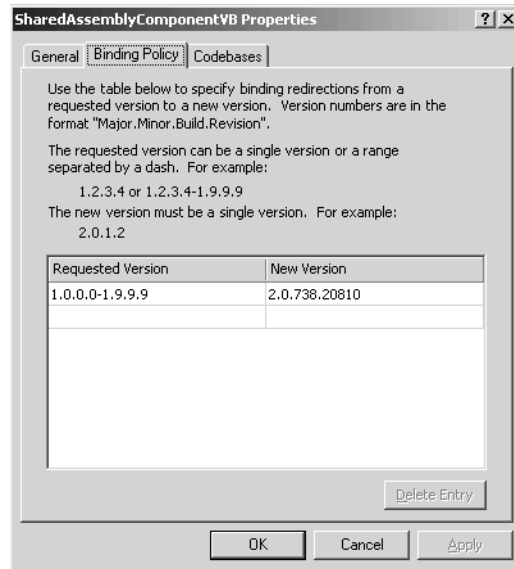


Figure 2-11 Setting binding policies.

An individual application can override a publisher policy's versioning behavior with its own configuration file.

If you need to redirect all the clients of one object version to a different version instead of leaving the original version on the machine for its original clients, then the machine-wide substitution that I just described is probably what you want. Occasionally, however, you might need to tell one old client to use a newer version of its server without changing the behavior of other old clients. You can do this with an application configuration file, which is an XML-based configuration file that modifies the behavior of a single application. The name of this file is the *full* name of the application that it configures, including the extension, with the additional extension ".config" tacked onto the end (e.g., SharedAssemblyClientVB.exe.config). This file lives in the client application's own directory. While masochists can produce it by hand, anyone who values her time will use the configuration utility on the client application itself. You add the client application to the Applications folder in the .NET Framework Configuration window. You must then add assemblies individually

to the application's Configured Assemblies section. You aren't moving the assemblies anywhere, you are simply adding configuration information to the local application's configuration file. The settings modify the default behavior of the assembly loader when accessed by this client only, even if the assembly lives in the GAC. When you set the configured assembly's properties, you'll see an option to allow you to ignore publisher policies. Selecting this option writes this information into your app configuration file, which will cause the loader to give you the app's original versioning behavior regardless of publisher policies. You can also specify a different version redirection, pointing your client app to someplace completely different. Just so you have an idea of what it looks like internally, Listing 2-5 shows the relevant portions of an application configuration file that ignore publisher policies and provide its own version redirection:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <publisherPolicy apply="no" />
      <dependentAssembly>
        <assemblyIdentity name="SharedAssemblyComponentVB"
          publicKeyToken="496ed8bd1d362eb2" />
        <publisherPolicy apply="no" />
        <bindingRedirect oldVersion="1.0.0.0-1.9.9.9"
          newVersion="2.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Listing 2-5 Sample configuration file.

Tips from the Trenches

My customers report that they often use the publisher policy to redirect all clients of a server version, but they almost never override the publisher policy with a private configuration file. "But what happens when we change the file but don't change the version number?" they often ask. One word: Don't. A new file means a new version. Stick to that.

Object-Oriented Programming Features

Organizing the internal functionality of software projects is difficult.

The only way to successfully develop software projects that require the work of more than a few developers is to partition the projects into classes of objects.

.NET provides all languages with the object-oriented features of inheritance and constructors.

When a software project reaches a certain level of complexity, the sheer effort of organizing the source code, of remembering the internal workings of every function, overwhelms the effort of dealing with your problem domain. No single person can remember what all the functions do and how they fit together, and chaos results. This critical size isn't very large, perhaps a five-programmer project, arguably less. To develop larger and more functional pieces of software—Microsoft Word for example—we need a better way of organizing code than providing global functions all over the place. Otherwise, the effort of picking our way through our spaghetti code overwhelms the work of figuring out how to process words.

The techniques of object-oriented programming were developed to solve this problem and allow larger, more complex programs to be developed. Exactly what someone means when he uses the term “object-oriented” is hard to pin down. The meaning depends heavily on the term's usage context and the shared background of the listeners. It's sort of like the word “love.” I once watched, amused, as two respected, relatively sober authors argued vehemently for half an hour over whether a particular programming technique truly deserved the description “object-oriented” or only the lesser “object-based.” But like love, most developers agree that object-oriented software is a Good Thing, even if they're somewhat vague on why and not completely sure about what. As most people will agree that the word “love,” at the minimum, indicates that you like something a lot, so most programmers will agree that object-oriented programming involves at least the partitioning of a program into *classes*, which combine logically related sets of data with the functions that act on that data. An *object* is an individual instance of a class. *Cat* is a class, my pet Simba is an instance of the class, an object. If you do a good job of segregating your program's functionality into classes that make sense, your developers don't have to understand the functionality of the entire program. They can concentrate on the particular class or classes involved in their subset of it, with (hopefully) minimal impact from other classes.

Providing object-oriented functionality to a programmer has historically been the job of the programming language, and different languages have taken it to different levels. Standard COBOL, for example, doesn't do it at all. Visual Basic provides a minimal degree of object-oriented functionality, essentially classes and nothing else. C++ and Java provide a high level of object-oriented features. Languages that want to work together seamlessly need to share the same degree of support for object-orientation, so the question facing Microsoft and developers was whether to smarten up Visual Basic

and other non-object-oriented languages or dumb down C++ and other languages that did support object-orientation. Because the architects of .NET belong (as do I) to the school of thought that says object-orientation is the only way to get anything useful done in the modern software industry, they decided that object-oriented features would be an integral part of the common language runtime environment, and thus available to all languages. The two most useful object-oriented techniques provided by the common language runtime are inheritance and constructors. I'll describe each of them in the following sections.

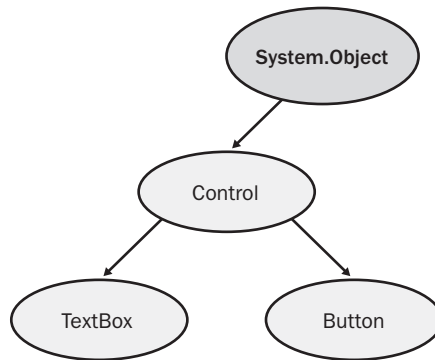
Inheritance

Essentially no manufacturer in modern industry, with the possible exception of glassmakers, builds their products entirely from nature, starting with earth, air, fire, and water. Instead, almost everyone reuses components that someone else has built, adding value in the process. For example, a company that sells camper trucks doesn't produce engine and chassis; instead they buy pickup trucks from an automaker and add specialized bodies to them. The automaker in turn bought the windshields from a glass manufacturer, who bought sand from a digger. We would like our software development process to follow this model, starting with generic functionality that someone else has already written and adding our own specialized attachments to it.

Essentially all modern economic processes involve adding value to existing components.

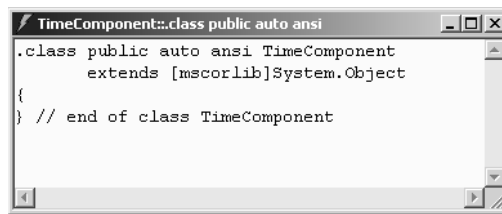
The object-oriented programming technique known as inheritance makes development of components much easier for programmers of software objects than it is for makers of physical objects. Someone somewhere uses a programming language that supports inheritance to write an object class, called the *base class*, which provides some useful generic functionality, say, reading and writing bytes from a stream. We'd like to use this basic functionality, with our own twists in it, in a class that reads and writes like the base class but that also provides statistics such as length. So we write a piece of software, known as the *derived class*, that incorporates the base class's functionality but modifies it in some manner, either adding more pieces to it, replacing some portion of it while leaving the rest intact, or a combination of both. We do this by simply telling the compiler that our derived class inherits from the base class, using the syntax of our programming language. The compiler will automatically include the base class's functionality in our derived class by reference. Think of it as cutting and pasting without actually moving anything. The derived class is said to *inherit from*, *derive from*, or *extend* the base class. The process is shown in Figure 2-12 with several intervening classes omitted for clarity.

Object-oriented programming provides this concept in software by means of inheritance.

**Figure 2-12** Object-oriented programming inheritance.

Every .NET object inherits from the system base class *System.Object*.

The .NET Framework uses inheritance to provide all kinds of system functionality, from the simplest string conversions to the most sophisticated Web services. To explore inheritance further, let's start as always with the simplest example we can find. The time component I wrote previously in this chapter offers a good illustration of .NET Framework inheritance. Even though I didn't explicitly write code to say so, our time component class derives from the Microsoft-provided base class *System.Object*. You can see that this is so by examining the component with ILDASM, as shown in Figure 2-13. All objects in the .NET system, without exception, derive from *System.Object* or another class that in turn derives from it. If you don't specify a different base class, *System.Object* is implied. If you prefer a different base class, you specify it by using the keyword *Inherits* in Visual Basic, as shown in Listing 2-6, or the colon operator in C#.

**Figure 2-13** ILDASM showing inheritance from *System.Object*.

```

Public Class WebService1
    Inherits System.Web.Services.WebService
  
```

Listing 2-6 Explicit declaration of inheritance.

In more complex cases, the Visual Studio .NET Object Browser shows us the inheritance tree. This case is too simple for it to handle. Figure 2-14 shows the Object Browser.

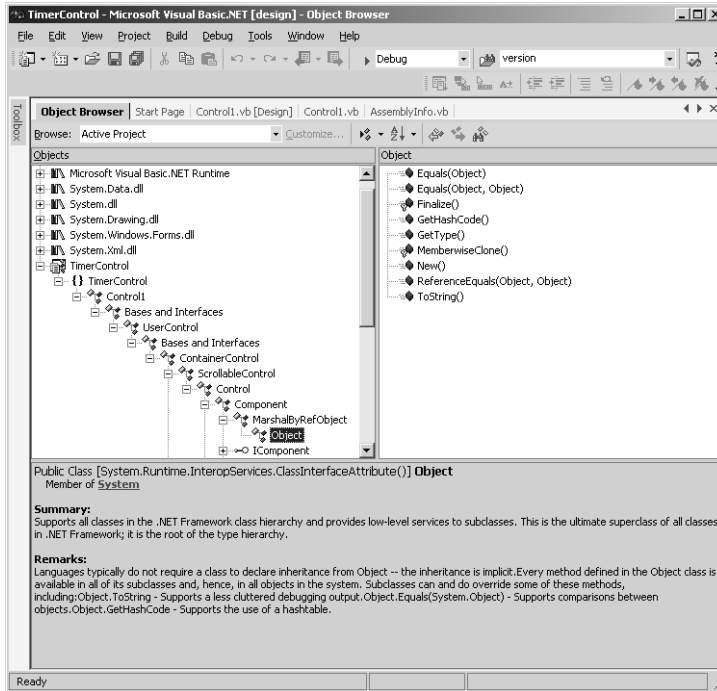


Figure 2-14 The Visual Studio Object Browser showing the inheritance tree.

OK, our time component inherits functionality from *System.Object*, but how do we know what was in the will? We find that out with a little old-fashioned RTFM (Read The Funny Manual, more or less). When we do that, we find that our base class has the public methods shown in Table 2-1. That means that our derived class, the time component, knows how to do these things even though we didn't write code for them.

Table 2-1 Public Methods of *System.Object*

Method name	Purpose
<i>Equals</i>	Determines whether this object is the same instance as a specified object.
<i>GetHashCode</i>	Quickly generates and returns an integer that can be used to identify this object in a hash table or other indexing scheme.
<i>GetType</i>	Returns the system metadata of the object.
<i>ToString</i>	Returns a string that provides the object's view of itself.

The *Equals* method determines whether two object references do or do not refer to the same physical instance of an object. This determination was surprisingly difficult to make in COM and could easily be broken by an incorrectly implemented server, but in .NET our objects inherit this functionality from the base class. I've written a client application that creates several instances of our time component and demonstrates the *Equals* method, among others. It's shown in Figure 2-15.

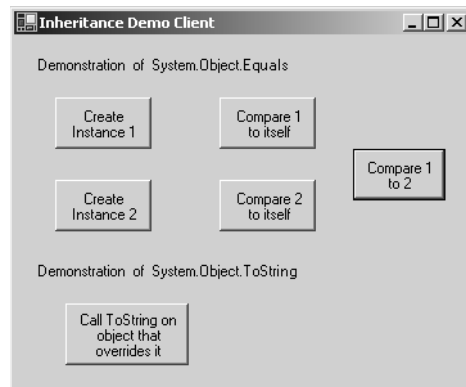


Figure 2-15 Client program demonstrating *System.Object* features inherited by time component.

You can override a base class's methods to replace part of its functionality.

Sometimes your component doesn't want everything it inherits from a base class, just like human heirs. You love the antique table your Aunt Sophie left you, but you aren't real crazy about her flatulent bulldog (or vice versa). Software inheritance generally allows a derived class to *override* a method that it inherits from the base class: that is, provide a replacement for it. A good example of this is the method *System.Object.ToString*, which tells an object to return a string for display to a programmer who is debugging the application. The implementation that we inherit from *System.Object* simply returns the name of the derived class, which isn't that illuminating. To make our component easier to debug, we'd like this method to return more detailed information. For example, an object that represents an open file might return the name of that file, or a *System.Boolean* object return its value "true" or "false". We do that by overriding the base class's method, as shown in Listing 2-7. We write a method in our derived class that has the same name and parameters as the method in the base class, specifying the keyword *Overrides* (*override* in C#) to tell the compiler to replace the base class's implementation with our derived class's new one.

```

' This method overrides the ToString method of the
' universal base class System.Object.

Public Overrides Function ToString() As String

    ' Call the base class's ToString method and get the result.
    ' You don't have to do this if you don't want to. I did,
    ' for demo purposes.

    Dim BaseResult As String
    BaseResult = MyBase.ToString

    ' Construct response string with base class's string plus
    ' my own added information. The net result here is that
    ' I'm piggybacking on the base class, not completely
    ' replacing it.

    Return "You have reached the overriding class. " + _
        "The base class says: " + BaseResult

End Function

```

Listing 2-7 Overriding base class method.

If your derived class wants to provide its own functionality in addition to that of the base class—rather than instead of the base class—it can call the overridden base class's method explicitly. In Visual Basic, the base class is accessible through the named object *MyBase*, and in C# it's called *base*. The sample component calls the base class to get its string and then appends its own string to that of the base class. The result is that the component is piggybacking on the base class's functionality rather than completely replacing it.

Most classes can serve as base classes for derivation. A few cannot, such as *System.String*. These classes are marked as *NotInheritable* in Visual Basic and *sealed* in C#. A class designer does this with classes that have fragile innards that he's worried another programmer might break. For example, the *String* class has been highly optimized behind the scenes to improve its performance, and its designers don't want to worry about breaking derived classes if they ever change this code. Other classes must serve only as base classes for derivation; you can't instantiate them directly. These are marked as *MustInherit* in Visual Basic and *abstract* in C#. An example of an abstract base class is *System.IO.Stream*. A designer makes a class abstract to force a common design pattern on a set of derived classes, by means of abstract methods described in the next paragraph.

An overriding method can access the base class's method that it overrides.

Some classes can't serve as bases, but others have to.

46 Introducing Microsoft .NET, Third Edition

Some methods can't be overridden, while others must be.

Most base class methods can be overridden, as I've shown you, but some of them can't and a few of them must be. A base class method must contain the keyword *Overridable* in Visual Basic or *virtual* in C# if you want to allow this. The method *System.Object.GetType* is not so written, and therefore cannot be overridden. The designers of the class hierarchy thought this class's functionality was too important for proper operation of many parts of the system to allow anyone to monkey with it. An abstract base class, on the other hand, contains methods that must be overridden, marked as *MustOverride* in Visual Basic or *abstract* in C#. For example, the abstract base class *System.IO.Stream* contains the abstract method *Read*. Classes that derive from it, such as *System.IO.MemoryStream* and *System.IO.FileStream*, must provide their own implementations of this method. Clients therefore see a common set of functions from all stream-derived classes.

.NET inheritance works between different languages.

Much is made of the ability of .NET to provide cross-language inheritance, that is, to allow a class written in one language, Visual Basic, for example, to derive from a base class written in another language, say C#. COM couldn't provide this feature because the differences between language implementations were too great. However, the standardized IL architecture of the common language runtime allows .NET applications to use it. In fact, the simple time component example does exactly that with no effort on my part. I guarantee you that the *System.Object* class is written in one language and not any other, yet every .NET object, without exception and regardless of language, inherits from it.

Object Constructors

Objects need a standard place for putting initialization code.

As the Good Rats sang a couple of decades ago, "birth comes to us all." As humans mark births with various rituals (religious observances, starting a college fund), so objects need a location where their birth-ritual code can be placed. Object-oriented programming has long recognized the concept of a *class constructor*, a function called when an object is created. (Object-oriented programming also uses the concept of a *class destructor*, a function called when the object is destroyed, but this concept has been replaced in .NET with the system garbage collector described in the next section.) Different languages have implemented constructors differently—C++ with the class name, Visual Basic with *Class_Initialize*. As with so many features that have varied widely among languages, the rituals for object creation had to be standardized for code written in different languages to work together properly.

.NET object classes provide for initialization through an object constructor.

In .NET, Visual Basic lost its *Class_Initialize* event, and the model looks much more like a C++ model, primarily because parameterized constructors are needed to support inheritance. Every .NET class can have one or more constructor methods. This method has the name *New* in Visual Basic .NET or

the class name in C#. The constructor function is called when a client creates your object using the *new* operator. In the function, you place the code that does whatever initialization your object requires, perhaps acquiring resources and setting them to their initial state. An example of a constructor is shown in Listing 2-8.

```
Public Class Point

    Public x, y As Integer

    ' Default constructor accepts no parameters,
    ' initializes member variables to zero

    Public Sub New()
        x = 0
        y = 0
    End Sub

    ' This constructor accepts two parameters, initializing
    ' member variables to the supplied values.

    Public Sub New(ByVal newx As Integer, ByVal newy As Integer)
        x = newx
        y = newy
    End Sub

End Class
```

Listing 2-8 Constructor declaration example.

One of the more interesting things you can do with a constructor is allow the client to pass parameters to it, thereby allowing the client to place the object in a particular state immediately upon its creation. For example, the constructor of an object representing a point on a graph might accept two integer values, the X and Y location of that point. You can even have several different constructors for your class that accept different sets of parameters. For example, our *Point* object class might have one constructor that accepts two values, another that accepts a single existing point, and yet a third that accepts no parameters and simply initializes the new point's members as zero. An example is shown in Listing 2-9. This flexibility is especially useful if you want to make an object that requires initialization before you can use it. Suppose you have an object that represents a patient in a hospital, supporting methods such as *Patient.ChargeLotsOfMoney* and *Patient.Amputate (which-Limb)*. Obviously, it is vital to know which human being each individual instance of this class refers to or you might remove money or limbs from the wrong patient, both of which are bad ideas, the latter generally more so than

Object constructors can accept different sets of parameters, allowing an object to be created in a particular state.

the former. By providing a constructor that requires a patient ID—and not providing a default empty constructor—you ensure that no one can ever operate on an unidentified patient or inadvertently change a patient's ID once it's created.

```
Dim foo As New Point ( )  
  
Dim bar As New Point (4, 5)
```

Listing 2-9 Constructor call example.

Tips from the Trenches

The technique of requiring a nondefault constructor breaks in several design cases. A .NET class that deserializes itself from XML (see Chapter 7) or a .NET class accessed by a COM client (see later in this chapter) requires a constructor that has no parameters. You can have as many more as you like, but you must have one constructor that takes no parameters.

.NET Memory Management

Manual memory management leads to costly, hard-to-find bugs.

One of the main sources of nasty, difficult-to-find bugs in modern applications is incorrect use of manual memory management. Older languages such as C++ required programmers to manually delete objects that they had created, which led to two main problems. First, programmers would create an object and forget to delete it when they finished using it. These leaks eventually consumed a process's entire memory space and caused it to crash. Second, programmers would manually delete an object but then mistakenly try to access its memory location later. Visual Basic would have detected the reference to invalid memory immediately, but C++ often doesn't. Sometimes the transistors that had made up the deleted object memory would still contain plausible values, and the program would continue to run with corrupted data. These mistakes seem painfully obvious in the trivial examples discussed here, and it's easy to say, "Well, just don't do that, you doofus." But in real programs, you often create an object in one part of the program and delete it in another, with complex logic intervening—logic deleting the object in some cases but not others. Both of these bugs are devilishly difficult to reproduce and harder still to track down. Programming discipline helps, of course, but we'd really like some way to keep our programmers thinking about our busi-

ness logic, not about resource management. You can bet that Julia Child, the *grand dame* of TV chefs, hires someone to clean up her kitchen when she's done with it so that she can concentrate on the parts of cooking that require her unique problem-domain expertise.

Modern languages such as Visual Basic and Java don't have this type of problem. These languages feature "fire-and-forget" automatic memory management, which is one of the main reasons that programmers select them for development. A Visual Basic 6.0 programmer doesn't have to remember to delete the objects that she creates in almost all cases. (Remember that "almost"; it will figure into an important design decision later.) Visual Basic 6.0 counts the references to each object and automatically deletes the object and reclaims its memory when its count reaches zero. Her development environment provides her with an automatic scullery maid cleaning the used pots and pans out of her sink and placing them back on her shelves. Wish I could get the same thing for my real kitchen. Maybe if you tell all your friends to buy this book....

Microsoft has made automatic memory management part of the .NET common language runtime, which allows it to be used from any language. It's conceptually simple, as shown in Figure 2-16.

Automatic memory management and resource recovery of the type built into Visual Basic and Java is a very useful feature.

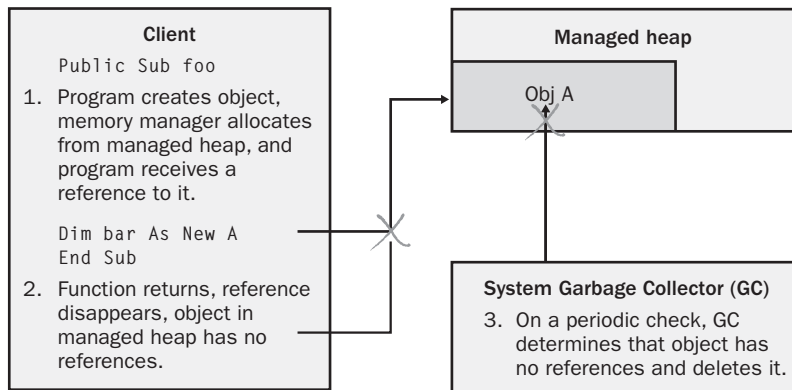


Figure 2-16 Automatic memory management with garbage collection.

A programmer creates an object using the *new* operator and receives a reference to it. The common language runtime allocates that object's memory from the *managed heap*, a portion of a process's memory reserved by the runtime for this purpose. Every so often, a system thread examines all the objects in the managed heap to see which of them the program still holds outstanding references to. An object to which all references have disappeared is called *garbage* and is removed from the managed heap. The objects remaining in the managed heap are then compacted together, and the exist-

The common language runtime garbage collector makes automatic memory management available to any application.

50 Introducing Microsoft .NET, Third Edition

ing references in the program fixed to point to their new location. The entire operation is called *garbage collection*. It solves the aforementioned problems of manual memory management without you having to write any code. You can't forget to delete an object because the system cleans up after you. And you can't access a deleted object through an invalid reference because the object won't be deleted as long as you hold any reference to it. Obviously, garbage collection is going to take more CPU cycles to run than just a standard in-out heap allocator, even though it is written to ensure that it doesn't check an object twice or get caught in circular object references. As I said previously, I think this is a good investment of CPU cycles because it gets you faster development time with fewer bugs.

The garbage collector runs when it feels like it, but you can force a garbage collection manually.

This magical collection of garbage takes place when the garbage collector darn well feels like it. Apart from detecting no more available memory in the managed heap in response to an allocation request, no one really knows what the exact algorithm is for launching a garbage collection, and I wouldn't be surprised to see it vary from one version to another of the released product. You can force a garbage collection manually by calling the function *System.GC.Collect*. You might want to make this call at logical points in your program; for example, to clear away the debris just after a user saves a file or perhaps to clear the decks just before starting a large operation. Most of the time you just let the garbage collector do its thing when it wants to.

Before garbage collection, we often put cleanup code in an object's destructor or *Class_Terminate* method.

Automatic garbage collection looks great so far, but it leaves us with one gaping hole. What about the cleanup that an object needs to do when it gets destroyed? C++ applications usually cleaned up in an object's destructor, and Visual Basic classes did the same thing in their *Class_Terminate* methods. This is a good location for cleanup code because a client can't forget to call it, but how can we handle this with automatic garbage collection? First, let's realize that the problem has gotten considerably smaller. The main cleanup task we performed in C++ destructors was to delete additional objects to which the destructing object held references, and now garbage collection takes care of that for us automatically. But occasionally we'll need to do some cleanup that doesn't involve local garbage-collected resources; for example, releasing a database connection or logging out from a remote system.

The garbage collector supports an object finalizer method for necessary cleanup code.

The common language runtime garbage collection supports the notion of a *finalizer*, an object method that is called when the object is garbage collected. It is somewhat analogous to a C++ class destructor and also to the Visual Basic *Class_Terminate* method, both of which it replaces. However, a finalizer is significantly different from both of these other mechanisms in ways you may find unsettling. The universal runtime base class *System.Object* contains a method called *Finalize*, which we override as shown in Listing 2-10. When the object is garbage collected, the garbage collection thread detects the fact that our object has a *Finalize* method and calls it, thereby executing our

cleanup code. Although early versions didn't do it, the released version of .NET calls all outstanding finalizers automatically when an application shuts down.

In C#, you supply a finalizer by writing what looks like an ordinary destructor, but under the hood your compiler is overriding the *Finalize* method and it behaves as a garbage-collected finalizer and not a deterministic destructor as in C++. This is the only case I've ever seen in which Visual Basic code provides a clearer view of what's really going on behind the scenes than a C-family language does.

```
Protected Overrides Sub Finalize()  
  
    ' Perform whatever finalization logic we need.  
  
    MessageBox.Show("In Finalize, my number = " + _  
        MyObjectNumber.ToString())  
  
    ' Forward the call to our base class.  
  
    MyBase.Finalize()  
  
End Sub
```

Listing 2-10 Providing a *Finalize* function in an object.

Note Finalizers look simple, but their internal behavior is actually quite complex and it's fairly easy to mess them up. If you are planning on using them, you **MUST** read Jeffrey Richter's account of garbage collection in his book, *Applied Microsoft .NET Framework Programming* (Microsoft Press, 2002). The fact that it took him a whole chapter to describe it should tell you something about the internal complexity of garbage collection, even if, or perhaps because, its connection to your program is so simple.

Using a finalizer has some disadvantages as well. Obviously it consumes CPU cycles, so you shouldn't use it if you have no cleanup to do. There is no way to guarantee the order in which the garbage collector calls the finalizers of garbage objects, so don't depend on one object finalizing before or after another, regardless of the order in which the last reference to each of them disappeared. Finalizers are called on a separate garbage-collector thread within your application, so you can't do any of your own serialization to enforce a calling order or you'll break the whole garbage collection system in

Using a finalizer can be trickier than it looks.

52 Introducing Microsoft .NET, Third Edition

your process. Since your object became garbage, the objects that it holds might have become garbage too unless you've taken steps to prevent that from happening. Don't plan on calling any other object in your application from your finalizer unless you've explicitly written code to ensure that someone is holding a reference to keep the other object from becoming garbage. Don't plan on throwing exceptions (see the section about structured exception handling later in this chapter) from your finalizer; no one is listening to you any more, you garbage object, you. And make sure you catch any exceptions generated by your cleanup code so that you don't disturb the garbage collector's thread that calls your finalizer.

Finalizers are fine if we don't care when our cleanup gets done, if "eventually, by the time you really need it, I promise" is soon enough. Sometimes this is OK, but it isn't so good if the resources that a finalizer would recover are scarce in the running process—database connections, for example. Eventual recovery isn't good enough; we need this object shredded NOW so that we can recover its expensive resources that the generic garbage collector doesn't know about. We could force an immediate garbage collection, as discussed previously, but that requires examining the entire managed heap, which can be quite expensive even if there's nothing else to clean up. Since we know exactly which object we want to dismantle, we'd like a way of cleaning up only that object, as Julia often wipes off her favorite paring knife without having to clean up her entire kitchen (including taking out the garbage). This operation goes by the grand name of *deterministic finalization*. Objects that want to support deterministic finalization do so by implementing an interface called *IDisposable*, which contains the single method, *Dispose*. In this method, you place whatever code you need to release your expensive resources. The client calls this method to tell the object to release those resources right now. For example, all Windows Forms objects that represent a window in the underlying operating system support this feature to enable quick recovery of the operating system window handle that they contain.

Sometimes you will see an object provide a different method name for deterministic finalization in order for the name to make sense to a developer who needs to figure out which method to call. For example, calling *Dispose* on a file object would make you think that you were shredding the file, so the developer of such an object will provide deterministic finalization through a method with the more logical name of *Close*.

Deterministic finalization sounds like a good idea, but it also contains its own drawbacks. You can't be sure that a client will remember to call your *Dispose* method, so you need to provide cleanup functionality in your finalizer as well. However, if your client does call *Dispose*, you probably don't

An object that wants to provide a deterministic way for a client to release its resources exposes a method called *Dispose*.

want the garbage collector to waste its time calling your object's finalizer, as the cleanup should have already been done by the *Dispose* method. By calling the function *System.GC.SuppressFinalize*, you tell the garbage collector not to bother calling your finalizer even though you have one. A Visual Basic object also needs to expressly forward the *Dispose* call to its base class if the base class contains a *Dispose* method, as the call won't otherwise get there and you will fail to release the base class's expensive resources. A C# destructor does this automatically. A sample *Dispose* method is shown in Listing 2-11. This class is derived from *System.Object*, which doesn't contain a *Dispose* method, so I've omitted the code that would forward that call.

I've written a small sample program that illustrates the concepts of automatic memory management and garbage collection. You can download it from this book's Web site. A picture of the client app is shown in Figure 2-17. Note that calling *Dispose* does not make an object garbage. In fact, by definition, you can't call *Dispose* on an object that is garbage because then you wouldn't have a reference with which to call *Dispose*. The object won't become garbage until no more references to it exist, whenever that may be. I'd suggest that your object maintain an internal flag to remember when it has been disposed of and to respond to any other access after its disposal by throwing an exception.

You have to write code to handle the case in which a client accesses your object after calling *Dispose* on it.

```
Public Class Class1
    Implements System.IDisposable

    Public Sub Dispose() Implements System.IDisposable.Dispose
        ' Do whatever logic we need to do to immediately free up
        ' our resources.

        MessageBox.Show("In Dispose(), my number = " + _
            MyObjectNumber.ToString())

        ' If our base class contained a Dispose method, we'd
        ' forward the call to it by uncommenting the following line.

        ' MyBase.Dispose()

        ' Mark our object as no longer needing finalization.

        System.GC.SuppressFinalize(Me)
    End Sub

End Class
```

Listing 2-11 Sample *Dispose* method for deterministic finalization.

54 Introducing Microsoft .NET, Third Edition

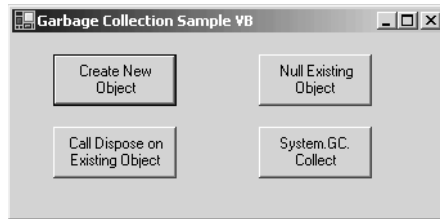


Figure 2-17 Memory management client application.

Microsoft decided on garbage collection memory management to make it leak proof, even at the cost of easy determinism.

While automatic garbage collection makes the simple operations of allocating and freeing objects easier to write and harder to mess up than they were in C++, it makes deterministic finalization harder to write and easier to mess up than it was in Visual Basic 6. C++ programmers will probably consider this a great profit, while Visual Basic programmers, who are used to their automatic, almost foolproof behavior also being deterministic, may at first consider it a step back. The reason that Microsoft switched to garbage collection is that Visual Basic's reference counting algorithm didn't correctly handle the case of circular object references, as in the case where a child object holds a reference to its parent. Suppose object A creates object B, object B creates object C, and object C obtains and holds a reference to its parent, object B. Suppose that object A now releases object B. Object B won't be destroyed now because object C still holds a reference to it, and C won't let go until B lets go of it. Unless a programmer writes code to break the circular reference before A lets go, both B and C are leaked away, orphans with no references except their hold on each other, which keeps them both alive. The garbage collection algorithm will automatically detect and handle this circular reference case, while reference counting will not. After much discussion of alternatives and banging of heads against walls, Microsoft decided that foolproof, automatic leak prevention in all cases was more important than easy determinism. Some programmers will agree, others won't, but the choice was carefully reasoned and not capricious. After an initial period of suspicion, I'm coming around to this way of thinking. I find that I don't need deterministic finalization very often, and as a refugee from C++ memory leaks, I REALLY love the fire-and-forget nature of garbage collection.

Tips from the Trenches

The adoption of garbage collection has gone pretty much as I expected. My customers report that they absolutely love its automatic fire-and-forget nature, but they hate the deterministic finalization design pattern because clients often forget to call *Dispose*. Remember, you need deterministic finalization only in two cases: where your object is a wrapper for an expensive resource, or where you need to enforce object cleanup in a certain order. A somewhat hacky solution to the former case is to place a static counter in the class that wraps expensive resources, increment that counter in the class's constructor, and, when it hits a certain value, reset it and force a garbage collection.

Interoperation with COM

The commercial success of any new software platform depends critically on how well it integrates with what already exists while providing new avenues for development of even better applications. For example, Windows 3.0 not only allowed existing DOS applications to run, but it also multitasked them better than any other product up to that point and provided a platform for writing Windows applications that were better than any DOS app. The canvas on which we paint is essentially never blank. How did God manage to create the world in only six days? He didn't have any installed base to worry about being backward compatible with. (My editor points out that He also skimped on documentation.)

Windows has depended on COM for interapplication communication since 1993. Essentially all code for the Windows environment is neck-deep in COM and has been for an awfully long time in geek years. The .NET Framework has to support COM to have any chance of succeeding commercially. And it does, both as a .NET client using a COM server, and vice versa. Since it is more likely that new .NET code will have to interoperate with existing COM code than the reverse, I will describe that case first.

Backward compatibility is crucial in the development of any new system. Therefore, .NET supports interoperation with COM.

Using COM Objects from .NET

A .NET client accesses a COM server by means of a *runtime callable wrapper* (RCW), as shown in Figure 2-18. The RCW wraps the COM object and mediates between it and the common language runtime environment, making the

A .NET client accesses a COM object through a runtime callable wrapper (RCW).

COM object appear to .NET clients just as if it were a native .NET object and making the .NET client appear to the COM object just as if it were a standard COM client.

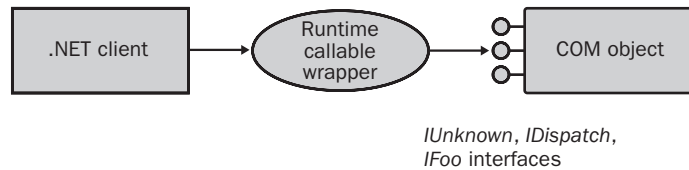


Figure 2-18 .NET client/COM object interaction via a runtime callable wrapper.

You can generate an RCW with a variety of development tools.

The developer of a .NET client generates the RCW in one of two ways. If you're using Visual Studio .NET, simply right-click on the References section of your project and select Add Reference from the context menu. You will see the dialog box shown in Figure 2-19, which offers a choice of all the registered COM objects it finds on the system. Select the COM object for which you want to generate the RCW, and Visual Studio .NET will spit it out for you. If you're not using Visual Studio .NET, the .NET SDK contains a command line tool called TlbImp.exe, the type library importer that performs the same task. The logic that reads the type library and generates the RCW code actually lives in a .NET run-time class called *System.Runtime.InteropServices.TypeLibConverter*. Both Visual Studio .NET and TlbImp.exe use this class internally, and you can too if you're writing a development tool or feeling masochistic.

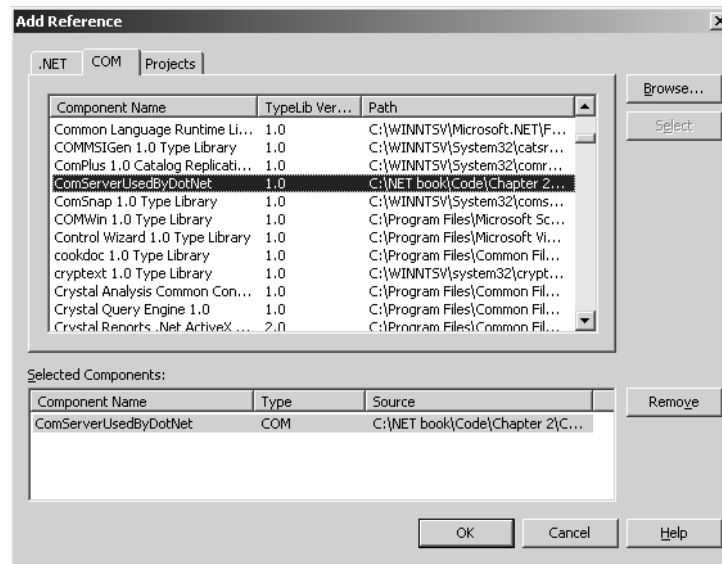


Figure 2-19 Locating COM objects for RCW generation.

Figure 2-20 shows a sample .NET client program that uses a COM object server. You can download the samples and follow along from the book's Web site. This sample contains a COM server, a COM client, and a .NET client so that you can compare the two. The source code is shown in Listing 2-12.

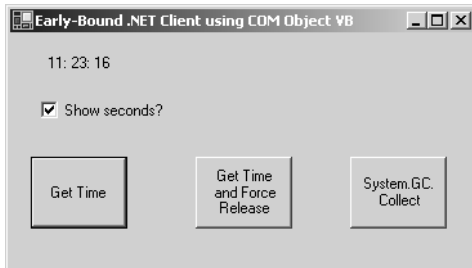


Figure 2-20 Sample .NET client using a COM server.

```
Protected Sub Button1_Click(ByVal sender As Object, _
                           ByVal e As System.EventArgs)

    ' Create an instance of the RCW that wraps our COM object.

    Dim RuntimeCallableWrapper As New ComUsedByDotNet.Class1()

    ' Call the method that gets the time.

    Label1.Text = RuntimeCallableWrapper.GetTimeFromCom(CheckBox1.Checked)

    ' Object becomes garbage when it goes out of scope,
    ' but is not actually released until next garbage collection.

End Sub
```

Listing 2-12 Code listing of a .NET client using an RCW.

After you generate the RCW as described in the preceding paragraph, you will probably want to import its namespace into the client program using the *Imports* statement, allowing you to refer to the object using its short name. You create the RCW object simply by using the *new* operator, as you would for any other .NET object. When it's created, the RCW internally calls the native COM function *CoCreateInstance*, thereby creating the COM object that it wraps. Your .NET client program then calls methods on the RCW as if it were a native .NET object. The RCW automatically converts each call to the COM calling convention—for example, converting .NET strings into the BSTR strings that COM requires—and forwards it to the object. The RCW converts the results returned from the COM object into native .NET types before

The RCW magically converts .NET calls into COM and COM results to .NET.

58 Introducing Microsoft .NET, Third Edition

COM objects are actually destroyed when their RCWs are garbage collected.

returning them to the client. Users of the COM support in Visual J++ will find this architecture familiar.

When you run the sample COM client program, you'll notice (from dialog boxes that I place in the code) that the object is created when you click the button and then immediately destroyed. When you run the sample .NET client program, you'll find that the object is created when you click the Get Time button, but that the object isn't destroyed immediately. You would think it should be, as the wrapper object goes out of scope, but it isn't, not even if you explicitly set the object reference to nothing. This is the .NET way of lazy resource recovery, described previously in the section about garbage collection. The RCW has gone out of scope and is no longer accessible to your program, but it doesn't actually release the COM object that it wraps until the RCW is garbage collected and destroyed. This can be a problem, as most COM objects were not written with this life cycle in mind and thus might retain expensive resources that should be released as soon as the client is finished. You can solve this problem in one of two ways. The first, obviously, is by forcing an immediate garbage collection via the function *System.GC.Collect*. Calling this function will collect and reclaim all system resources that are no longer in use, including all the RCWs not currently in scope. The drawback to this approach is that the overhead of a full garbage collection can be high, and you may not want to pay it immediately just to shred one object. If you would like to blow away one particular COM object without affecting the others, you can do so via the function *System.Runtime.InteropServices.Marshal.ReleaseComObject*.

The RCW mechanism described in the preceding paragraphs requires an object to be early-bound, by which I mean that the developer must have intimate knowledge of the object at development time to construct the wrapper class. Not all objects work this way. For example, scripting situations require late binding, in which a client reads the ProgID of an object and the method to call on it from script code at run time. Most COM objects support the *IDispatch* interface specifically to allow this type of late-bound access. Creating an RCW in advance is not possible in situations like this. Can .NET also handle it?

.NET also supports late binding without too much trouble.

Fortunately, it can. The .NET Framework supports late binding to the *IDispatch* interface supported by most COM objects. A sample late binding program is shown in Figure 2-21, and its code in Listing 2-13. You create a system type based on the object's ProgID via the static method *Type.GetTypeFromProgID*. The static method *Type.GetTypeFromCLSID* (not shown) does the same thing based on a CLSID, if you have that instead of a ProgID. Once you have the type, you create the COM object using the

method *Activator.CreateInstance* and call a method via the function *Type.InvokeMember*. These functions are part of .NET reflection, which I discuss in Chapter 11. It's more work—late binding always is—but you can do it.

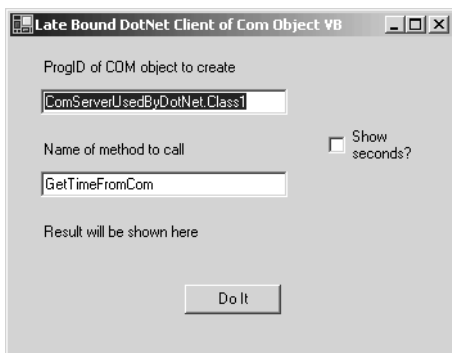


Figure 2-21 Sample late binding program.

```
Protected Sub Button1_Click(ByVal sender As Object, _
                             ByVal e As System.EventArgs)

    ' Get system type name based on prog ID.

    Dim MyType As System.Type
    MyType = Type.GetTypeFromProgID(textBox1().Text)

    ' Use an activator to create object of that type.

    Dim MyObj As Object
    MyObj = Activator.CreateInstance(MyType)

    ' Assemble array of parameters to pass to COM object.

    Dim prms() As Object = {checkBox1().Checked}

    ' Call method on object by its name.

    label2().Text = MyType.InvokeMember("GetTimeFromCom", _
        Reflection.BindingFlags.InvokeMethod, Nothing, MyObj, _
        prms).ToString()

End Sub
```

Listing 2-13 Sample late binding code.

Tips from the Trenches

Clients report from the field that they don't like using COM from their .NET programs. The code required to work with COM is easy to write, but the inclusion of even one badly behaved COM object can bring down an otherwise robust .NET application. For example, COM objects don't allocate memory from the managed heap, even when you access them through an RCW from a .NET application. They therefore can leak memory that the .NET garbage collector can't reclaim. The interoperation capability lets you develop test cases and pilot projects and use third party COM components whose vendors haven't yet ported them to .NET. But if you use this capability as a long-term solution, you'll be missing many of the benefits of .NET.

Using .NET Objects from COM

A COM client accesses a .NET object through a COM callable wrapper (CCW).

Suppose, on the other hand, you have a client that already speaks COM and now you want to make it use a .NET object instead. This is a somewhat less common scenario than the reverse situation that I've previously described because it presupposes new COM development in a .NET world. But I can easily see it occurring in the situation in which you have an existing client that uses 10 COM objects and you now want to add an 11th set of functionality that exists only as a .NET object—and you want all of them to look the same to the client for consistency. The .NET Framework supports this situation as well, by means of a *COM callable wrapper* (CCW), as shown in Figure 2-22. The CCW wraps up the .NET object and mediates between it and the common language runtime environment, making the .NET object appear to COM clients just as if it were a native COM object.

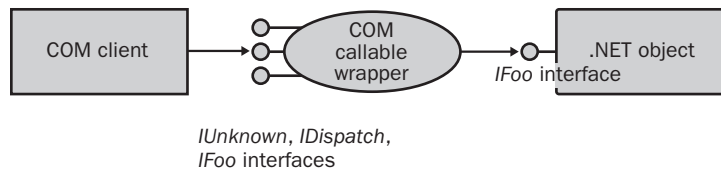


Figure 2-22 COM callable wrapper.

A .NET component must be signed, live in the GAC, and provide a default constructor to work with a COM client.

To operate with a COM callable wrapper, a .NET component's assembly must be signed with a strong name; otherwise the common language runtime won't be able to definitively identify it. It must also reside in the GAC or, less commonly, in the client application's directory. However, as

was the case previously when building the shared component's client, the component must also reside at registration time in a standard directory outside the GAC. Any .NET class that you want COM to create must provide a default constructor, by which I mean a constructor that requires no parameters. COM object creation functions don't know how to pass parameters to the objects that they create, so you need to make sure your class doesn't require this. Your class can have as many parameterized constructors as you want for the use of .NET clients, as long as you have one that requires none for the use of COM clients.

For a COM client to find the .NET object, we need to make the registry entries that COM requires. You do this with a utility program, called *RegAsm.exe*, that comes with the .NET Framework SDK. This program reads the metadata in a .NET class and makes registry entries that point the COM client to it. The sample code provides a batch file that does this for you. The registry entries that it makes are shown in Figure 2-23. Notice that the COM server for this operation is the intermediary DLL *Mscoree.dll*. The *Class* value of the *InProcServer32* key tells this DLL which .NET class to create and wrap, and the *Assembly* entry tells it in which assembly it will find this class.

The SDK utility *RegAsm.exe* makes registry entries telling COM where to find the server for the .NET class.

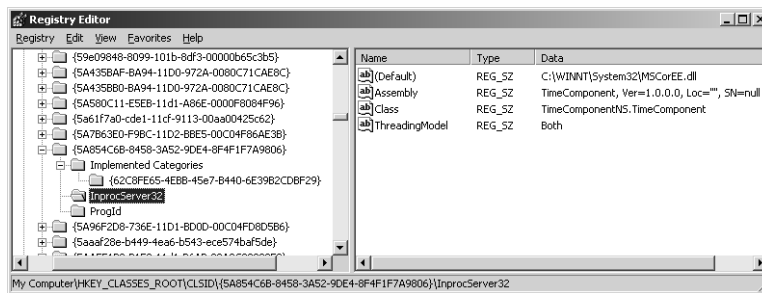


Figure 2-23 Registry entries made by *RegAsm.exe*.

A COM client accesses a .NET object as if it were a native COM object. When the client calls *CoCreateInstance* to create the object, the registry directs the request to the registered server, *Mscoree.dll*. This DLL inspects the requested CLSID, reads the registry to find the .NET class to create, and rolls a CCW on the fly based on that .NET class. The CCW converts native COM types to their .NET equivalents—for example, BSTRs to .NET *Strings*—and forwards them to the .NET object. It also converts the results back from .NET into COM, including any errors. The sample code for this chapter contains a COM client that accesses the shared time component assembly that we built previously in this chapter.

The sample code for this chapter contains a COM client using a .NET object.

A .NET developer could reasonably want some methods, interfaces, or classes to be available to COM clients and others not to be. Therefore, .NET provides a metadata attribute called *System.Runtime.InteropServices.ComVis-*

ibleAttribute. (The .NET Framework allows you to use the abbreviated form of attribute class names, in this case *ComVisible* rather than *ComVisibleAttribute*. I'll be using the short version from now on.) You can use this attribute on an assembly, a class, an interface, or an individual method. Items marked with this attribute set to False will not be visible to COM. The default common language runtime setting is True, so the absence of this attribute causes the item to be visible to COM. However, the Visual Studio .NET default behavior for assemblies is to set this attribute's value to False in the AssemblyInfo.vb file. Settings made lower in the hierarchy override those made higher up. In the sample program, I set this attribute to True on my class, thereby making it visible to COM, as shown in the code that follows. If I wanted everything in the assembly visible to COM, I'd change it in AssemblyInfo.vb.

```
<System.Runtime.InteropServices.ComVisible(True)> Public Class Class1
```

Transactions in .NET

Transactions ensure the integrity of databases during complex operations.

Transactions are necessary to protect the integrity of data in distributed systems. Suppose we're writing an on-line bill paying application. Paying my phone bill requires us to debit my account in some database and credit the phone company's account, probably in a different database and possibly on a different machine. If the debit succeeds but the credit somehow fails, we need to undo the debit, or money would be destroyed and the integrity of the data in the system violated. We need to ensure that either both of these operations succeed or both of them fail. Performing both operations within a transaction does exactly that. If both operations succeed, the transaction commits and the new account values are saved. If either operation fails, the transaction aborts and all account values are rolled back to their original values. (To learn more about transactions in general, I highly recommend *Principles of Transaction Processing* by Philip A. Bernstein and Eric Newcomer, published by Morgan Kaufmann, 1997.)

COM+ contains good automatic transaction support.

COM+, and its ancestor, Microsoft Transaction Server (MTS), provided automatic support that made it easy for programmers to write objects that participated in transactions. A programmer marked his objects administratively as requiring a transaction. COM+ then automatically created one when the object was activated. The object used COM+ Resource Managers, programs such as Microsoft SQL Server that support the COM+ way of performing transactions, to make changes to a database. The object then told COM+ whether it was happy with the results. If all the objects participating in a transaction were happy, COM+ committed the transaction, telling the Resource Managers to save all their changes. If any object was unhappy, COM+ aborted the trans-

action, telling the Resource Managers to discard the results of all objects' operations, rolling back the state of the system to its original values. To learn more about COM+'s implementation of transactions, read my book *Understanding COM+* (Microsoft Press, 1999).

Native .NET objects can also participate in COM+ transactions. The .NET Framework contains a layer of code that mediates between COM+ and native .NET objects, encapsulated in the base class *System.EnterpriseServices.ServicedComponent*. Objects that want to participate in COM+ transactions (or use any other COM+ services, for that matter) must inherit from this class. Just as with .NET components accessed from COM, your .NET class must contain a default constructor (one that accepts no parameters), and you must sign it with a strong name. You specify your component's use of transactions by marking the class with the attribute *System.EnterpriseServices.Transaction*. The code in Listing 2-14 shows a class from my sample program that demonstrates COM+ transactions in a native .NET object.

Native .NET objects can participate in COM+ transactions by using pre-fabricated .NET Framework functionality.

```
Imports System.EnterpriseServices

' Mark our class as requiring a transaction

<Transaction(TransactionOption.Required)> Public Class Class1

    ' We need to inherit from ServicedComponent, which contains
    ' code for interacting with COM+
    Inherits ServicedComponent
    ' Mark this method to use .NET's automatic
    ' transaction voting (optional) <AutoComplete()> Sub AutoCompleteMethod
    ()
        (program logic omitted)
    End Sub
End Class
```

Listing 2-14 COM+ transactions in a native .NET object.

Your .NET client creates a transactional object using the *new* operator, exactly as for any other .NET object. When you do this, the *ServicedComponent* base class from which your object inherits first looks to see if the component is registered with the COM+ catalog. If it's not, the base class registers the component with the COM+ catalog, creating a COM+ application for it and adding the .NET class to it as a COM+ component, as shown in Figure 2-24. The metadata in the .NET class specifies its transactional requirements, which the base class uses to set the .NET component's properties in the COM+ catalog. The base class sets not only the component's use of transactions, but also the other COM+ properties implied by the transaction, such as JIT activation

The system base class registers your .NET component in the COM+ catalog.

64 Introducing Microsoft .NET, Third Edition

and synchronization. If you want your component to set these or other COM+ properties explicitly, you will find that the namespace *System.EnterpriseServices* contains many other attributes, such as *ObjectPooling*, with which you decorate your class to specify its behavior. Once your object is created, your client calls methods on it exactly as for any other .NET object. You can watch the sample program's transactional operations in Component Services, as shown in Figure 2-25.

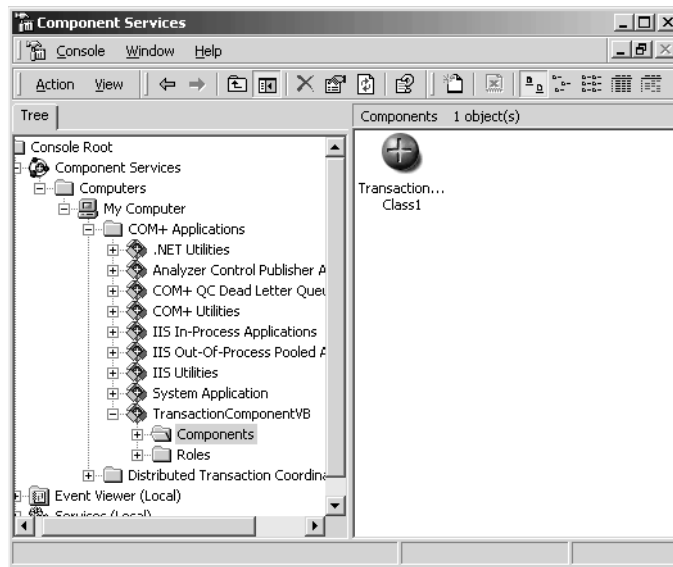


Figure 2-24 Transactional .NET component installed in Component Services Explorer.

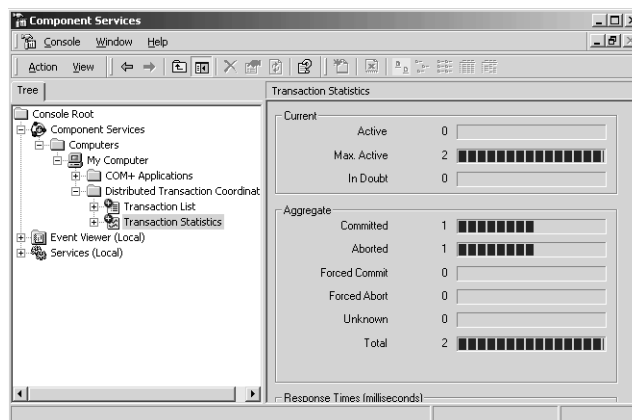


Figure 2-25 .NET component committing and aborting transactions.

An object that participates in a transaction needs to vote on that transaction's outcome. Your .NET transactional object can do this in one of two ways. The easiest way is to use the automatic transaction voting in .NET. You mark your transactional-component method with the attribute *System.EnterpriseServices.AutoComplete*, as shown previously in Listing 2-14. In this case, a method that returns without throwing an exception automatically calls *SetComplete* internally, while a method that throws an exception to its caller automatically calls *SetAbort* internally. If your methods are self-contained (as they should be anyway for good transactional component design), and if they signal failure to their clients by means of exceptions (as they should do anyway for good .NET component design), this approach is probably the best choice.

Alternatively, your .NET object might want to vote on the outcome of its transaction by means of explicit function calls. In COM+ and MTS, an object fetched its context by calling the API function *GetObjectContext* and then called a method on the context to indicate its transaction vote. A .NET object will find its context on the system-provided object named *System.EnterpriseServices.ContextUtil*. This object provides the commonly used methods *SetAbort* and *SetComplete*, and their somewhat less common siblings, *EnableCommit* and *DisableCommit*. These methods set your object's happiness and doneness bits in exactly the same manner as they did in COM+. The context also contains everything else you would expect to find in a COM+ context, such as the properties *DeactivateOnReturn* and *MyTransactionVote*, which allow you to read and set these bits individually. If your object requires several function calls to accomplish its work within a single transaction, or you haven't yet converted your error handling code to use structured exception handling, this choice is probably best for you.

A .NET transactional component can be configured so that it votes on its transaction automatically by throwing or not throwing an exception.

A .NET transactional component can also vote on its transaction via explicit function calls.

Structured Exception Handling

Every program encounters errors during its run time. The program tries to do something—open a file or create an object, for example—and the operation fails for one reason or another. How does your program find out whether an operation succeeded or failed, and how do you write code to handle the latter case?

The classic approach employed by a failed function is to return a special case value that indicated that failure, say, *Nothing* (or *NULL* in C++). This approach had three drawbacks. First, the programmer had to write code that checked the function's return value, and this often didn't happen in the time crunch that colors modern software development. Like seat belts or birth control, error-indicating return values only work if you use them. Errors didn't get trapped at their source but instead got propagated to higher levels of the

Every program needs to handle errors that occur at run time.

Returning a special case value to indicate the failure of a function doesn't work well.

program. There they were much more difficult to unravel and sometimes got masked until after a program shipped. Second, even if you were paying attention to it, the value of the error return code varied widely from one function to another, increasing the potential for programming mistakes. *CreateWindow*, for example, indicates a failure by returning NULL, *CreateFile* returns -1, and in 16-bit Windows, *LoadLibrary* returned any value less than 32. To make things even more chaotic, all COM-related functions return 0 as a success code and a nonzero value to indicate different types of failures. Third, a function could return only a single value to its caller, which didn't give a debugger (human or machine) very much information to work with in trying to understand and fix the error.

No other technique works well across languages either.

Different languages tried other approaches to handling run-time errors. Visual Basic used the *On Error GoTo* mechanism, which was and is a god-awful kludge. *GoTo* has no place in modern software; it hasn't for at least a decade and maybe more. C++ and Java used a better mechanism, called *structured exception handling* (SEH), which uses an object to carry information about a failure and a handler code block to deal with that object. Unfortunately, like most features of any pre-common language runtime language, structured exception handling only worked within that particular language. COM tried to provide rich, cross-language exception handling through the *ISupportErrorInfo* and *IErrorInfo* interfaces, but this approach was difficult to program and you were never sure whether your counterpart was following the same rules you were.

.NET provides structured exception handling as a fundamental feature available in and between all languages.

The .NET common language runtime provides structured exception handling, similar to that in C++ or Java, as a fundamental feature available to all languages. This architecture solves many of the problems that have dogged error handling in the past. An unhandled exception will shut down your application, so you can't ignore one during development. A function that is reporting a failure places its descriptive information in a .NET object, so it can contain any amount of information you'd like to report. Since the infrastructure is built into the runtime, you have to write very little code to take advantage of it. And as with all runtime functionality, .NET structured exception handling works well across all languages.

I've written a sample program that demonstrates some of the structured exception handling features in the common language runtime. Figure 2-26 shows a picture of it. You can download the code from the book's Web site and work along with me.

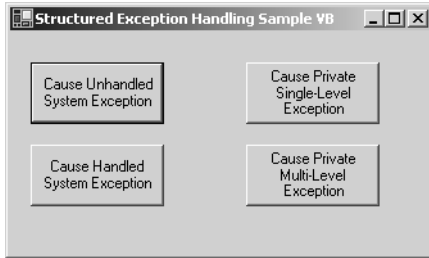


Figure 2-26 Sample program demonstrating structured exception handling.

A client program about to perform an operation that it thinks might fail sets up an *exception handler* block in its code, using the keywords *Try* and *Catch*, as shown in the Visual Basic .NET code in Listing 2-15. The exact syntax of structured exception handling varies from one language to another, but all the ones I've seen so far are pretty close to this.

A client program uses a *Try-Catch* block to specify its exception handling code.

```
Protected Sub btnHandled_Click(ByVal sender As Object, _
                               ByVal e As System.EventArgs)

    ' Entering this block of code writes an exception handler onto
    ' the stack.

    Try

        ' Perform an operation that we know will cause an exception.

        Dim foo As System.IO.FileStream
        foo = System.IO.File.Open("Non-existent file", IO.FileMode.Open)
        ' When an exception is thrown at a lower level of
        ' code, this handler block catches it.

    Catch x As System.Exception

        ' Perform whatever cleanup we want to do in response
        ' to the exception that we caught.

        MessageBox.Show(x.Message)
    End Try
End Sub
```

Listing 2-15 Client application code showing structured exception handling.

When program execution enters the *Try* block, the common language runtime writes an exception handler to the stack, as shown in Figure 2-27. When a called function lower down on the stack throws an exception, as described in the next paragraph, the runtime exception-handling mechanism starts examining the stack upward until it finds an exception handler. The stack is then unwound (all objects on it discarded), and control transfers to the exception handler. An exception can come from any depth in the call stack. In the sample program, I deliberately open a file that I know does not exist. The system method *File.Open* throws an exception, and my client catches it and displays information to the user about what has happened.

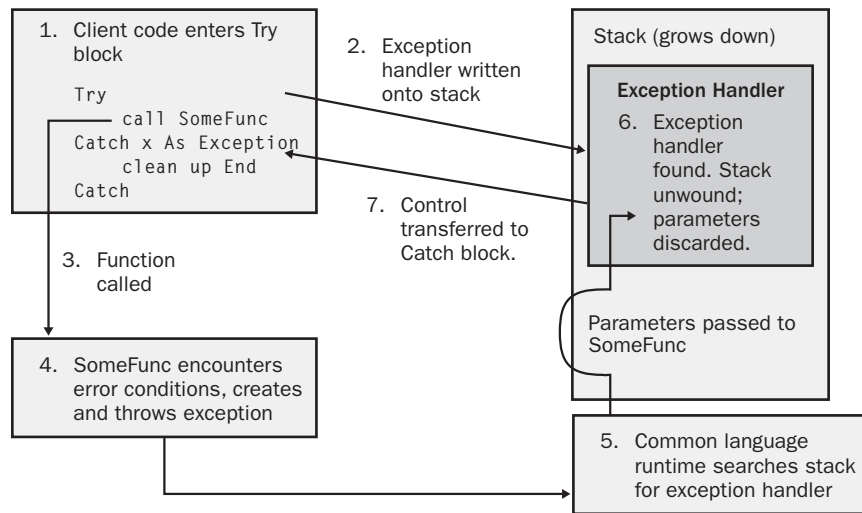


Figure 2-27 Structured exception handling diagram.

A piece of code that wants to throw an exception creates a *System.Exception* object, fills out its fields, and calls the system function *Throw*.

Any code that wants to can throw an exception. The common language runtime uses SEH for all of its error reporting, as shown in the previous example. For consistency, you therefore probably want to use SEH to signal errors from one part of your application to another. A piece of code that wants to throw an exception creates a new object of type *System.Exception*. You set the properties of this object to whatever you want them to be to describe the exception situation to any interested catchers. The common language runtime automatically includes a stack trace so that the exception handler code can tell exactly where the exception originated. Then you throw the exception using the keyword *Throw*, as shown in the code in Listing 2-16. This call tells the system to start examining the stack for handlers. The exception handler can live any number of levels above the exception thrower in the call stack.

```

Public Function BottomFunction() As String

    ' Create a new Exception object, setting its "Message" property,
    ' which can only be done in the constructor.

    Dim MyException _
        As New Exception("Exception thrown by BottomFunction")

    ' Set the new Exception's Source property, which can be
    ' done anywhere.

    MyException.Source = _
        "Introducing Microsoft .NET Chapter 2 ExceptionComponent"

    ' Throw the exception.

    Throw MyException

End Function

```

Listing 2-16 Throwing an exception in SEH.

When the common language runtime transfers control to an exception handler, the program stack between the thrower and the handler is discarded, as shown previously in Figure 2-27. Any objects or object references that existed on that stack become garbage. Because of the .NET automatic garbage collection, you don't have to worry about objects being leaked away, which was a big concern when using C++ native exception handling. However, having the objects discarded in this manner means that you don't get a chance to call the *Dispose* methods of any that needed deterministic finalization. Their finalizers will be called eventually at the next garbage collection, but that might not be soon enough. You can handle this situation with a *Try-Finally* handler, as shown in Listing 2-17. Code in a *Finally* block is executed as the stack is unwound, so you can put your cleanup code there. You can use both a *Catch* and a *Finally* block on the same *Try* if you want to.

SEH becomes even more powerful if throwers throw different types of exceptions to indicate different types of program failure. You do this by deriving your own class from the generic base class *System.Exception*. You can add any additional methods or properties to your exception class that you think would explain the situation to any potential catchers. In the example shown at the start of this section, when I attempted to open the nonexistent file, the system threw an exception of type *FileNotFoundException*, which contained the name of the file that it couldn't find. Even if you don't add anything else, the mere presence of a particular type of exception will indicate what type of failure has taken place. I wrote the handler shown in Listing 2-15 to catch any

You can enforce cleanup from an exception using a *Try-Finally* block.

You can throw and catch many different types of exceptions.

type of exception. If I wanted the handler to catch only exceptions of the type *FileNotFoundException*, I would change *Catch x As System.Exception* to *Catch x As System.IO.FileNotFoundException*. The common language runtime, when examining the stack, matches the type of exception thrown to the type specified in the *Catch* block, transferring control only if the type thrown matches exactly or is derived from the specified *Catch* type. A *Try* block can have any number of *Catch* handlers attached to it. The common language runtime will search them in the order in which they appear, so you want to put the most specific ones first.

```
Public Function MiddleFunction() As String

    ' Entering this block causes a handler to be written onto the stack.

    Try
        BottomFunction()

        ' The code in this Finally handler is executed whenever
        ' execution leaves the Try block for any reason. We care most
        ' about the case in which BottomFunction throws an exception
        ' and the stack is unwound. Without the Finally handler, we'd
        ' have no chance to clean up from that exception.

    Finally
        MessageBox.Show("Finally handler in MiddleFunction")
    End Try

End Function
```

Listing 2-17 Finally handler in structured error handling.

Tips from the Trenches

My customers report that they get their best use of exception handling when their code in any given place catches only the types of exceptions that it knows how to handle. Exception handling code shouldn't merely absorb all *System.Exception* objects as this sample does. However, they like putting a handler for all exceptions at the very top of their program, enclosing the *Application.Run* call in their *Main* function. An exception reaching that handler means that it somehow escaped all the lower handlers and would otherwise cause the program to crash. In this handler, they log the exception to a file or send e-mail, save any open work as best they can, and terminate peacefully. The stack trace in the exception shows exactly where it originated, which makes it much easier to fix.

Code Access Security

At the beginning of the PC era, very few users installed and ran code that they hadn't purchased from a store. The fact that a manufacturer had gotten shelf space at CompUSA or the late Egghead Software pretty much assured a customer that the software in the box didn't contain a malicious virus, as no nefarious schemer could afford that much marketing overhead. And, like Tylenol, the shrink-wrap on the package ensured a customer that it hadn't been tampered with since the manufacturer shipped it. While the software could and probably did have bugs that would occasionally cause problems, you were fairly comfortable that it wouldn't demolish your unbacked-up hard drive just for the pleasure of hearing you scream.

Customers generally feel that software purchased from a store is safe for them to run.

This security model doesn't work well today because most software doesn't come from a store any more. You install some large packages, like Microsoft Office or Visual Studio, from a CD, although I wonder how much longer even that will last as high-speed Internet connections proliferate. But what about updates to, say, Internet Explorer? A new game based on Tetris? Vendors love distributing software over the Web because it's cheaper and easier than cramming it through a retail channel, and consumers like it for the convenience and lower prices. And Web code isn't limited to what you've conventionally thought of as a software application. Web pages contain scripts that do various things, not all of them good. Even Office documents that people send you by e-mail can contain scripting macros. Numerically, except for perhaps the operating system, your computer probably contains more code functions that you downloaded from the Web than you installed from a CD you purchased, and the ratio is only going to increase.

However, most software today now comes from the Web.

While distributing software over the Web is great from an entrepreneurial standpoint, it raises security problems that we haven't had before. It's now much easier for a malicious person to spread evil through viruses. It seems that not a month goes by without some new virus alert on CNN, so the problem is obviously bad enough to regularly attract the attention of mainstream media. Security experts tell you to run only code sent by people you know well, but who else is an e-mail virus going to propagate to? And how can we try software from companies we've never heard of? It is essentially impossible for a user to know when code downloaded from the Web is safe and when it isn't. Even trusted and knowledgeable users can damage systems when they run malicious or buggy software. You could clamp down and not let your users run any code that your IT department hasn't personally installed. Try it for a day and see how much work you get done. We've become dependent on Web code to a degree you won't believe until you try to live without it. The only thing that's kept society as we know it from collapsing is the relative

It is essentially impossible for a user to know when code from the Web is safe and when it isn't.

72 Introducing Microsoft .NET, Third Edition

The Authenticode system doesn't protect you from harm; it merely identifies the person harming you.

We want to specify the levels of privilege that individual pieces of code can have, as we do with the humans in our lives.

The .NET common language runtime provides code access security at run time on a per-assembly basis.

scarcity of people with the combination of malicious inclination and technical skills to cause trouble.

Microsoft's first attempt to make Web code safe was its Authenticode system, introduced with the ActiveX SDK in 1996. Authenticode allowed manufacturers to attach a digital signature to downloaded controls so that the user would have some degree of certainty that the control really was coming from the person who said it was and that it hadn't been tampered with since it was signed. Authenticode worked fairly well to guarantee that the latest proposed update to Internet Explorer really did come from Microsoft and not some malicious spoofer. But Microsoft tried to reproduce the security conditions present in a retail store, not realizing that wasn't sufficient in a modern Internet world. The cursory examination required to get a digital certificate didn't assure a purchaser that a vendor wasn't malicious (like idiots, VeriSign gave *me* one, for only \$20), as the presence of a vendor's product on a store shelf or a mail-order catalog more or less did. Worst of all, Authenticode was an all-or-nothing deal. It told you with some degree of certainty who the code came from, but your only choice was to install it or not. Once the code was on your system, there was no way to keep it from harming you. Authenticode isn't a security system; it's an accountability system. It doesn't keep code from harming you, it just ensures that you know who to kill if it does.

What we really want is a way to restrict the operations that individual pieces of code can perform on the basis of the level of trust that we have in that code. You allow different people in your life to have different levels of access to your resources according to your level of trust in them: a (current) spouse can borrow your credit card; a friend can borrow your older car; a neighbor can borrow your garden hose. We want our operating system to support the same sort of distinctions. For example, we might want the operating system to enforce a restriction that a control we download from the Internet can access our user interface but can't access files on our disk, unless it comes from a small set of vendors who we've learned to trust. The Win32 operating system didn't support this type of functionality, as it wasn't originally designed for this purpose. But now we're in the arms of the common language runtime, which is.

The .NET common language runtime provides *code access security*, which allows an administrator to specify the privileges that each managed code assembly has, based on our degree of trust, if any, in that assembly. When managed code makes a runtime call to access a protected resource—

say, opening a file or accessing Active Directory—the runtime checks to see whether the administrator has granted that privilege to that assembly, as shown in Figure 2-28. The common language runtime walks all the way to the top of the call stack when performing this check so that an untrusted top-level assembly can't bypass the security system by employing trusted henchmen lower down. (If a nun attempts to pick your daughter up from school, you still want the teacher to check that you sent her, right?) Even though this checking slows down access to a protected resource, there's no other good way to avoid leaving a security hole. While the common language runtime can't govern the actions of unmanaged code, such as a COM object, which deals directly with the Win32 operating system instead of going through the runtime, the privilege of accessing unmanaged code can be granted or denied by the administrator.

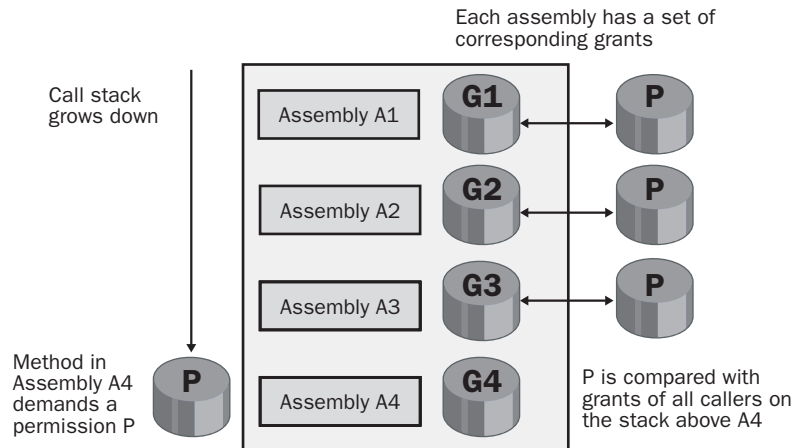


Figure 2-28 Access check in common language runtime code access security.

The administrator sets the *security policy*, a configurable set of rules that says which assemblies are and which aren't allowed to perform which types of operations. These permissions can be set at three levels: enterprise, machine, and user. A lower-level setting can tighten restrictions placed by settings at a higher level, but not loosen them. For example, if the machine-level permission allows a particular assembly to open a file, a user-level permission can deny the assembly that privilege, but not grant it.

74 Introducing Microsoft .NET, Third Edition

The administrator sets the code access security policy by editing XML-based configuration files.

An administrator sets the security policy by editing XML-based configuration files stored on a machine's disk. Any program that can modify an XML file can edit these files, so installation scripts are easy to write. Human administrators and developers will want to use the .NET Framework Configuration utility program `mscorcfg.msc`. This utility hadn't yet appeared when I wrote the first edition of this book, and I said some rather strong things about the lack of and the crying need for such a tool. Fortunately it now exists, whether because of what I wrote or despite it—or whether the development team even read it—I'm not sure. You can see the location of the security configuration files themselves in Figure 2-29.

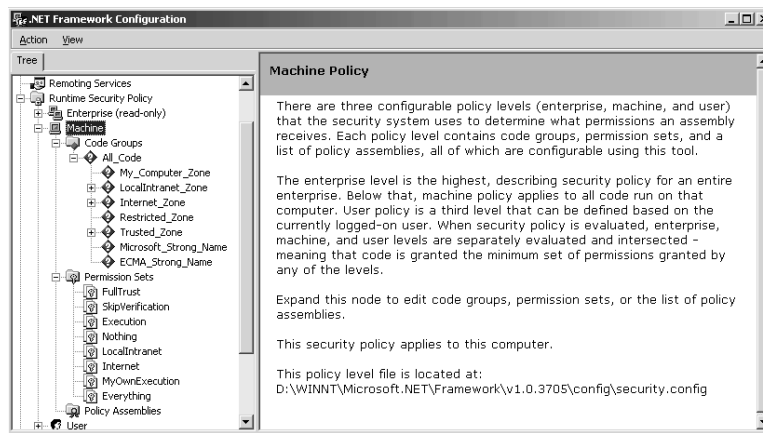


Figure 2-29 Security configuration files in the configuration utility.

The administrator constructs permission sets—lists of privileges that are granted and revoked as a group.

Rather than grant individual permissions to various applications, an administrator creates permission sets. These are (for once) exactly what their name implies—lists of things that you are allowed to do that can be granted or revoked as a unit to an assembly. The .NET Framework contains a built-in selection of permission sets, running from Full Trust (an assembly is allowed to do anything at all) to Nothing (an assembly is forbidden to do anything at all, including run), with several intermediate steps at permission levels that Microsoft thought users would find handy. The configuration tool prevents you from modifying the preconfigured sets, but it does allow you to make copies and modify the copies.

Each permission set consists of zero or more *permissions*. A permission is the right to use a logical subdivision of system functionality, for example, File Dialog or Environment Variables. Figure 2-30 shows the configuration tool dialog box allowing you to add or remove a permission to a permission set.

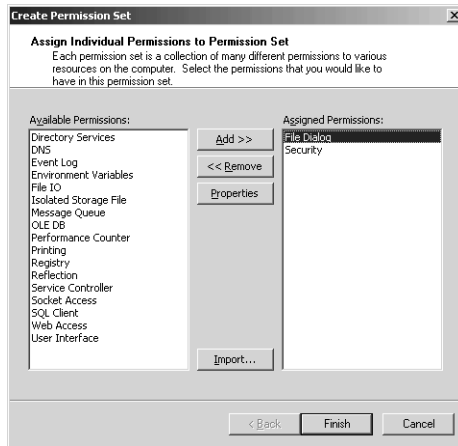


Figure 2-30 Assigning permissions to a permission set.

Each permission in turn supports one or more *properties*, which are finer-grained levels of permitted functionality that can be granted or revoked. For example, the File Dialog permission contains properties that allow an administrator to grant access to the Open dialog, the Save dialog, both, or neither. Figures 2-31 and 2-32 show the dialog boxes for setting the properties of the File Dialog permission and the ambiguously-named Security permission.

A permission set contains permissions, and a permission contains finer-grained properties.



Figure 2-31 Setting properties of a single permission.

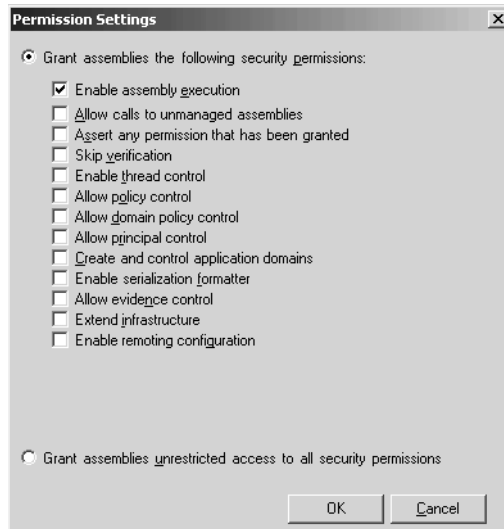


Figure 2-32 Setting properties of a different permission.

An administrator assigns assemblies to various code groups based on membership conditions such as where the code came from and whose digital signature it contains.

Now that you understand permission sets, let's look at how the administrator assigns a permission set to an assembly. A code-privilege administrator can assign a permission set to a specific assembly just as a log-in administrator can assign specific log-in privileges to an individual user. Both of these techniques, however, become unwieldy very quickly in production environments. Most log-in administrators set up user groups (data entry workers, officers, auditors, and so on), the members of which share a common level of privilege, and move individual users into and out of the groups. In a similar manner, most code-privilege administrators will set up groups of assemblies, known as *code groups*, and assign permission sets to these groups. The main difference between a log-in administrator's task and a code-privilege administrator's task is that the former will deal with each user's group membership manually, as new users come onto the system infrequently. Because of the way code is downloaded from the Web, we can't rely on a human to make a trust decision every time our browser encounters a new code assembly. A code-privilege administrator, therefore, sets up rules, known as *membership conditions*, that determine how assemblies are assigned to the various code groups. A membership condition will usually include the program zone that an assembly came from, for example, My Computer, Internet, Intranet, and so on. A membership condition can also include such information as the strong name of the assembly ("our developers wrote it"), or the public key with which it can be deciphered (and hence the private key with which it must have been signed). You set membership criteria using the .NET Framework Configuration tool, setting the properties of a code group, as shown in Figure 2-33.

Once you've set your membership criteria, you choose a permission set for members of that group, as shown in Figure 2-34. By default, .NET provides full trust to any assembly signed with Microsoft's strong name. I will be curious to see how many customers like this setting and how many do not.

The administrator then assigns a permission set to each code group.

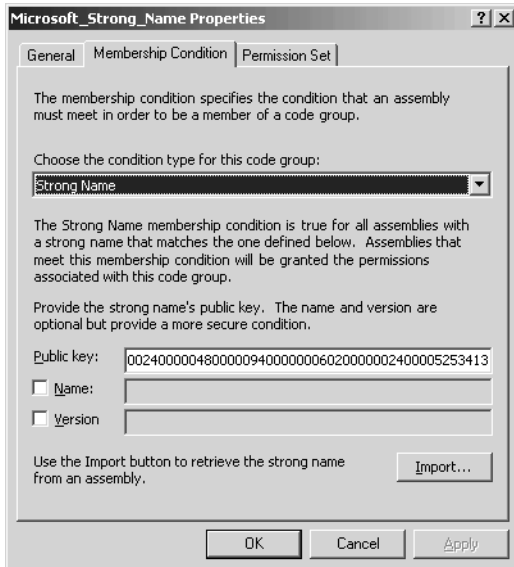


Figure 2-33 Setting code group membership conditions.

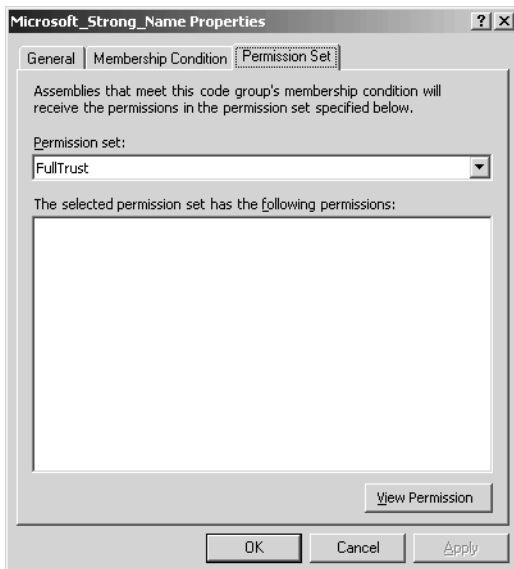


Figure 2-34 Assigning a permission set to a code group.

78 Introducing Microsoft .NET, Third Edition

If an assembly belongs to more than one code group, its permission set is the sum of the permission sets of the groups to which it belongs.

The common language runtime contains many functions and objects for interaction with the code-access security system programmatically.

When the common language runtime loads an assembly at run time, it figures out which code group the assembly belongs to by checking its membership conditions. It is common for an application to belong to more than one code group. For example, if I ran Microsoft Money add-ons from the Web, they would belong to both the Microsoft group (because I can decode its signature with Microsoft's public key) and the Internet Zone group (because I ran them from the Internet). When code belongs to more than one group, the permission sets for each group are added together, and the resulting (generally larger) permission set is applied to the application.

While most of the effort involved in code access security falls on system administrators, programmers will occasionally need to write code that deals with the code-access security system. For example, a programmer might want to add metadata attributes to an assembly specifying the permission set that it needs to get its work done. This doesn't affect the permission level it will get, as that's controlled by the administrative settings I've just described, but it will allow the common language runtime to fail its load immediately instead of waiting for the assembly to try an operation that it's not allowed to do. A programmer might also want to read the level of permission that an assembly actually has been granted so that the program can inform the user or an administrator what it's missing. The common language runtime contains many functions and objects that allow programmers to write code that interacts with the code-access security system. Even a cursory examination of these functions and objects is far beyond the scope of this book, but you should know that they exist, that you can work with them if you want to, and that you almost never will want to. If you set the administrative permissions the way I've just described, the right assemblies will be able to do the right things, and the wrong assemblies will be barred from doing the wrong things, and anyone who has time to write code for micromanaging operations within those criteria is welcome to.