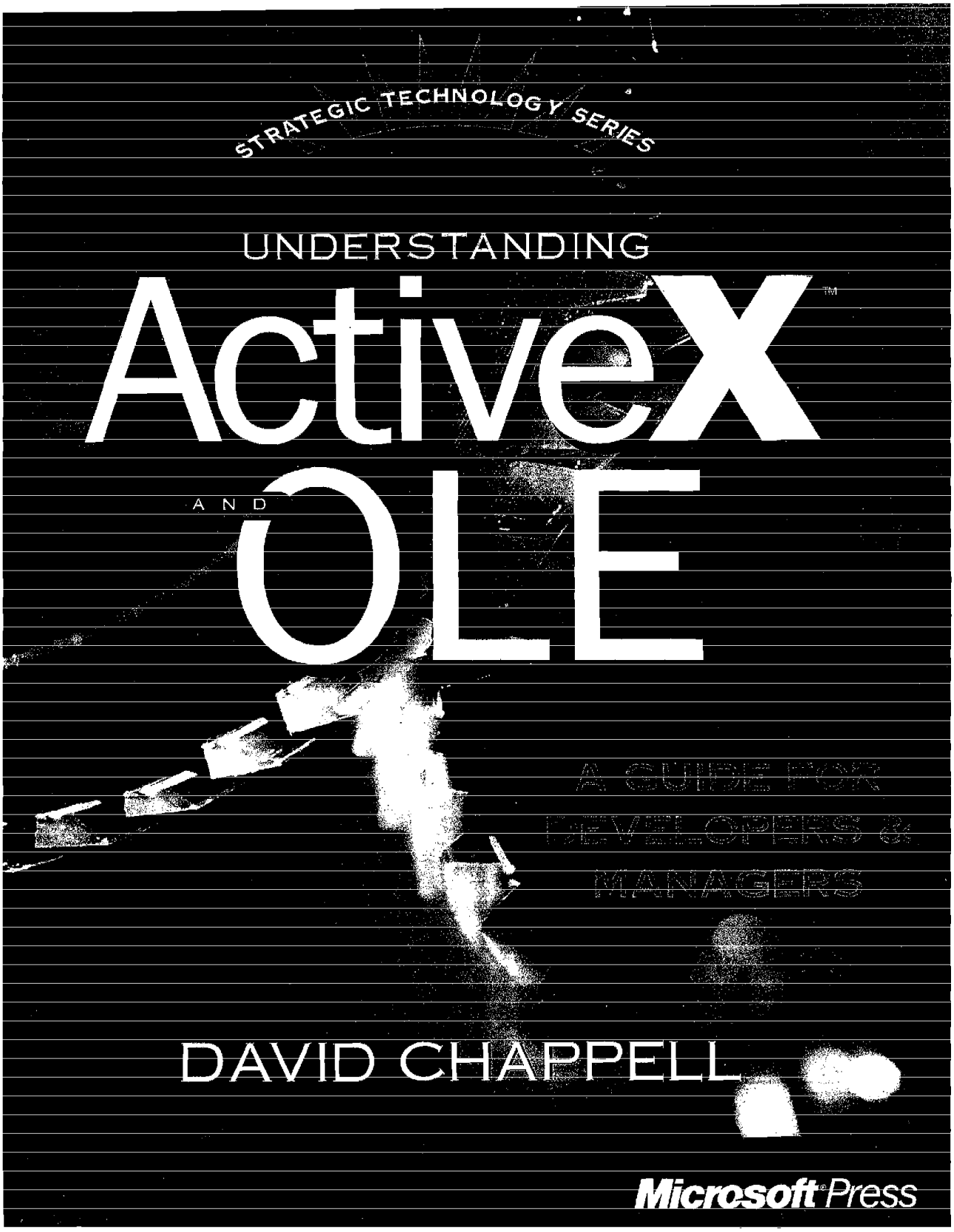

STRATEGIC TECHNOLOGY SERIES
UNDERSTANDING
ActiveX™
AND
OLE
A GUIDE FOR
DEVELOPERS &
MANAGERS
DAVID CHAPPELL
Microsoft Press

Understanding ActiveX and OLE

Published by **Microsoft Press**
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data pending.

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 QMQM 1 0 9 8 7 6

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7329.

Acquisitions Editor: **David Clark**
Project Editors: **Stuart J. Stuple and Mary Renaud**
Technical Editor: **Stuart J. Stuple**
Manuscript Editor: **Mary Renaud**

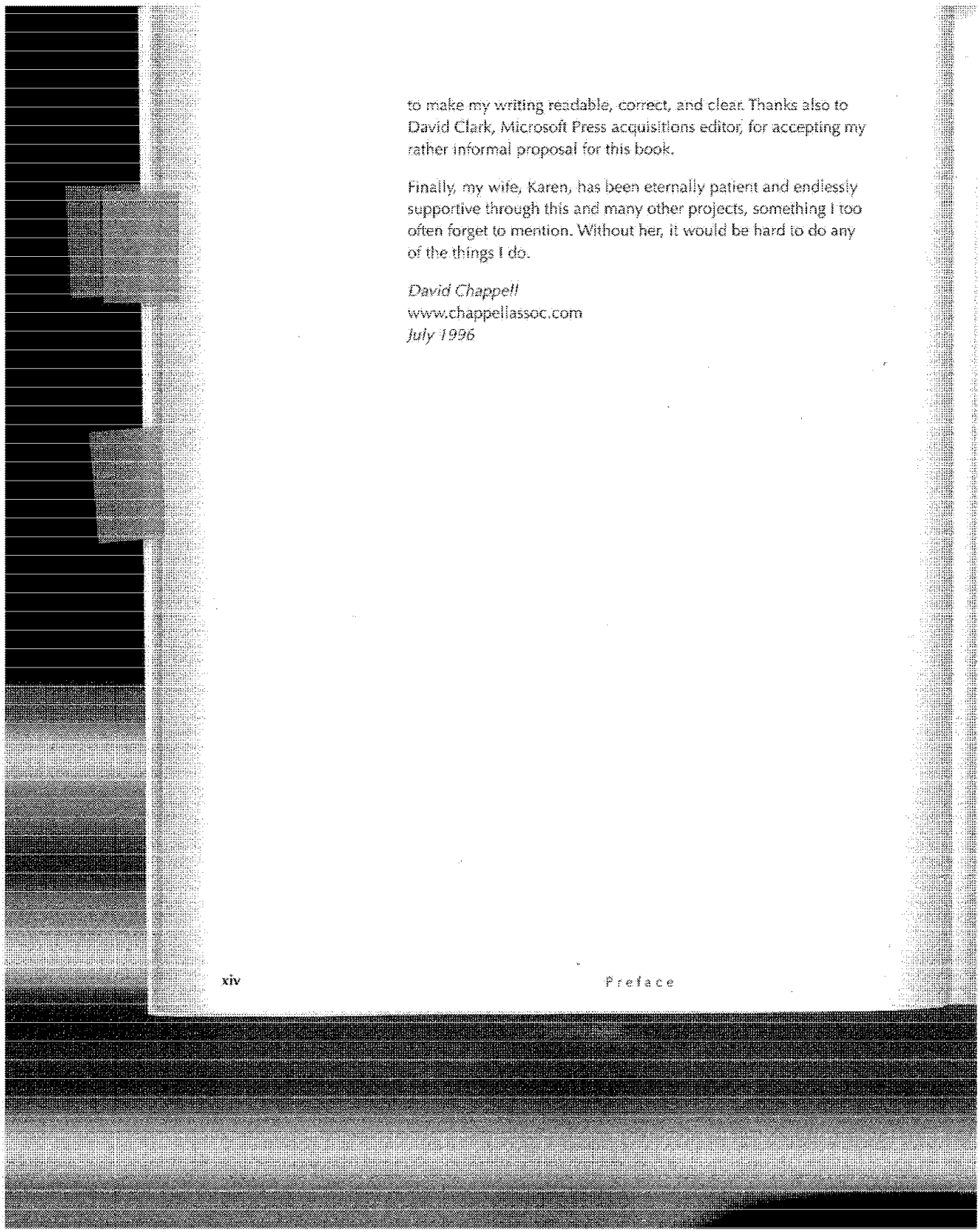
want to use ActiveX and OLE technologies in the software they develop, but it encompasses a broader audience as well. As you can quickly determine by flipping through the pages, this is not a programming book—it contains almost no code. Although I do assume that the reader is a software professional of some kind, I do not assume knowledge of C++ or Windows-based programming. ActiveX, OLE, and COM are important, and knowing what they are and how they work matters to a broader group than those who program for Microsoft Windows. Some familiarity with using Windows is taken for granted, however; this seemed safe to me, as it's hard to find anyone in this field who hasn't used Windows at least a little.

A Timestamp

The ActiveX and OLE technologies are a moving target. This book describes the fundamental COM-based technologies as of mid-1996. In particular, Chapter 10 on Distributed COM and Chapter 11 on the ActiveX Internet-related technologies were completed before those technologies actually shipped. Accordingly, some details described in these chapters might not exactly match what is finally delivered.

Where to Find More Detail

For some people, the depth of coverage offered in this book will be enough. (For others, it will surely be too much.) Developers who need a more intimate understanding of the topic will want to get a copy of the OLE "bible" for programmers, Kraig Brockschmidt's *Inside OLE*, 2d ed. (Microsoft Press, 1995). Another useful book, one that covers an important topic that's not fully addressed in *Inside OLE*, is *OLE Controls Inside Out*, by Adam Denning (Microsoft Press, 1995). (Watch for a new edition of this book, too, one that describes the recent changes in what are now known as ActiveX controls.) For the truly hard-core



to make my writing readable, correct, and clear. Thanks also to David Clark, Microsoft Press acquisitions editor, for accepting my rather informal proposal for this book.

Finally, my wife, Karen, has been eternally patient and endlessly supportive through this and many other projects, something I too often forget to mention. Without her, it would be hard to do any of the things I do.

David Chappell
www.chappellassoc.com
July 1996

Chapter One

Introducing ActiveX and OLE

Writing good software is hard. Writing software that's large and complex, as most code is today, is even harder. As computers continue to infiltrate our lives, as we depend on them for everything from running our cars to writing letters to making toast, the effectiveness and reliability of software become more and more important. Good code is becoming the bedrock of our civilization.

Writing good software is just plain hard

In some ways, the history of software is the history of efforts to write better code. Applications and system software both have suffered from endless delays, mind-boggling complexity, and more bugs than anyone cares to admit. But creating software is tough—there's no way around it. Doing it well requires the ability to take a big-picture view coupled with a willingness (an eagerness, even) to deal with a myriad of small details. The intellectual effort required is substantial, and the tools are never perfect.

ActiveX and OLE are about writing better software

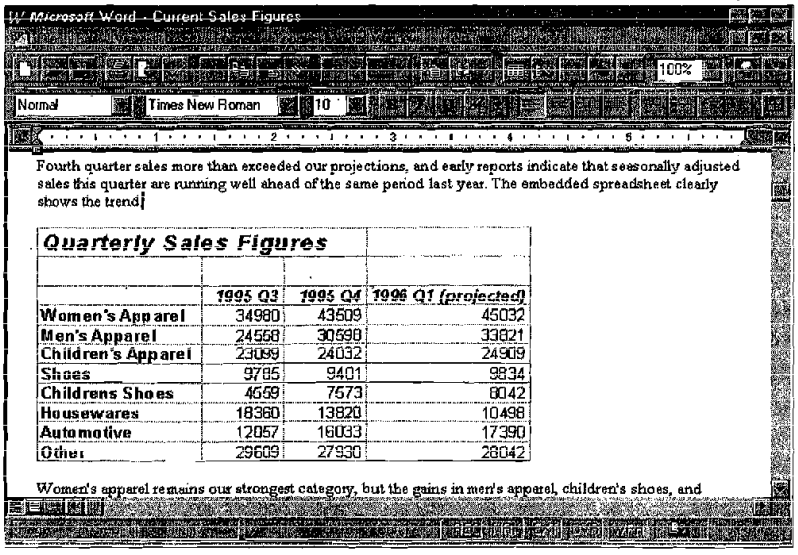
Microsoft's ActiveX and OLE are a step toward the creation of better software. "Better" here means software that's more reliable, certainly, and more effective as well. But it also means software that can do things that were impossible before, software that enables solutions to new problems. Although ActiveX and OLE are built on a quite simple idea, this idea turns out to have profound implications for improving how we create software.

From OLE to ActiveX

OLE 1 provided a way to create compound documents

The first incarnation of OLE, Object Linking and Embedding 1, was a mechanism for creating and working with compound documents. To its user, a compound document appears to be a single set of information, but in fact it contains elements created by two or more different applications. With OLE 1, for example, a user could combine a spreadsheet created using Microsoft Excel with a text document created using Microsoft Word, as shown in Figure 1-1. The idea was to give users a "document-centric" view of computing, to let them think more about their information and less about the applications they were using to work with that information. As the name suggested, compound documents could be created either by linking two separate documents together or by completely embedding one document in another.

Figure 1-1 ***A user's view of a compound document.***



OLE 2 introduced the Component Object Model

Like most version 1 software releases, OLE 1 wasn't perfect. The architects of the next release set out to improve on the original design. They soon realized that the compound-document problem was actually a special case of a more general problem: how

should various software components provide services to one another? To address this larger problem, OLE's architects created a set of technologies that were applicable to much more than compound documents. Foremost among these technologies was the Component Object Model (COM), which provided the foundation for OLE 2. This new version of OLE supported compound documents even better than the first release, but clearly a lot more was going on here than simply combining documents created by different applications. OLE 2 offered the potential for a new way of thinking about how software of all kinds should interact.

COM is a foundation for interaction among all kinds of software

This potential was largely the result of COM. COM establishes a common paradigm for interaction among all sorts of software—libraries, applications, system software, and more. Accordingly, virtually any kind of software technology can be implemented using the approach COM defines, and doing so offers some very tangible benefits.

The name Object Linking and Embedding became simply OLE

Because of those benefits, COM soon became a part of technologies that had nothing to do with compound documents. Microsoft, however, still wanted to have a common name to refer to all COM-based technologies as a group. The company decided to reduce the name *Object Linking and Embedding* to just *OLE*—this three-letter combination was no longer treated as an acronym—and to drop the version number.

The OLE label was applied to any technology that used COM

Under this new regime, the term *OLE* was applied to anything built using the paradigm COM provides (although COM was also used in products that didn't have *OLE* in their name). OLE no longer meant only compound documents but was now a label assigned to any COM-based technology. In some ways, grouping under a single name all software written using COM makes no more sense than, say, grouping together all software written in C++. Both COM and a programming language such as C++ are general tools that can be used to create all kinds of software. Still, both for historical reasons and to mark the advent of this new and far-reaching technology, the term *OLE* was used to identify many (but not quite all) COM-based technologies.

Today, most COM-based technologies are assigned the label ActiveX

In early 1996, Microsoft dropped another term into the fray: *ActiveX*. In its first appearances, this new term was associated with technologies related to the Internet and applications that grew out of the Internet, such as the World Wide Web. Because most of Microsoft's efforts in this area were based on COM, ActiveX was directly connected to OLE. Soon, though, this new term began to usurp more and more of OLE's traditional territory, and today things have come full circle. Now the term *OLE* once again refers only to the technology used to create compound documents through Object Linking and Embedding. The diverse set of technologies built using COM, once all grouped under the OLE label, are now grouped under the ActiveX banner. In several cases, technologies that had *OLE* in their name have been rechristened as ActiveX technologies. New COM-based technologies that once might have been given the OLE label are now frequently tagged with ActiveX instead.

Is this the end of the naming saga for COM-based technologies? Given the history so far, the answer is probably no. What Microsoft's marketing mavens will think up next is anybody's guess. But despite these adventures in nomenclature, what's really important hasn't changed. What's really important is COM.

Understanding COM

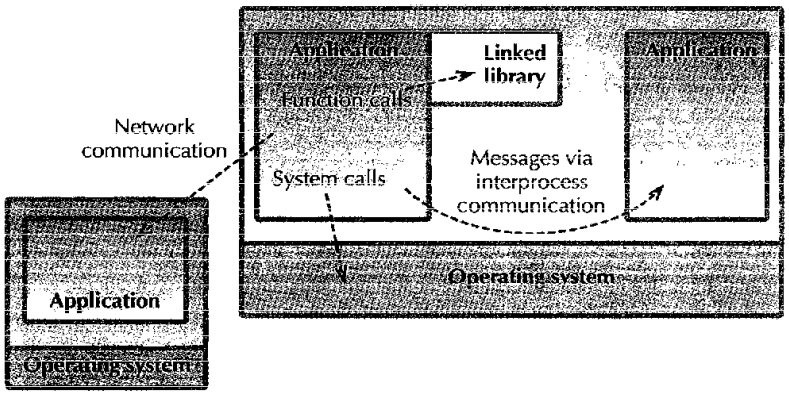
Traditionally, different kinds of software provided services in different ways

All OLE technologies and all the ActiveX technologies described in this book are built on the foundation provided by COM. So just what is COM? To answer this question, think first about another: how should one chunk of software access the services provided by another chunk of software? Today, as shown in Figure 1-2, the answer depends on what those chunks of software are. An application might, for example, link to a library and then access the library's services by calling the functions in the library. Or one application might use the services provided by another, which runs in an entirely separate process. In this case, the two local processes typically communicate by using an interprocess communication mechanism, which usually requires defining a *protocol* between the two applications (a set of messages allowing one

application to specify its requests and the other to respond appropriately). A third example is an application that might use services provided by an operating system. Here the application commonly makes system calls, each of which is handled by the operating system. Or, finally, an application might need the services of software that is running on a completely different machine, accessible via a network. Many different approaches can be used to access these services, such as exchanging messages with the remote application or issuing remote procedure calls.

Without COM, different mechanisms are used to access the services provided by libraries, local processes, the operating system, and remote processes.

Figure 1-2



The fundamental need in all these relationships is the same: one chunk of software must access services provided by another. But the mechanism for getting at those services differs in each case—local function calls, messages passed via interprocess communication, system calls (which in fact look pretty much like function calls to the programmer), or some kind of network communication. Why is this? Wouldn't it be simpler to define one common way to access all kinds of software services, regardless of how they are provided?

Accessing services in different ways is needlessly complex

This is exactly what COM does. It defines a standard approach by which one chunk of software supplies its services to another, an approach that works in all the cases just described. By applying

COM defines a common way to access software services

this common service architecture across libraries, applications, system software, and networks, COM is transforming the way software is constructed.

How COM Works

COM objects provide services via methods that are grouped into interfaces

With COM, any chunk of software implements its services as one or more *COM objects*.[1] Every COM object supports one or more *interfaces*, each of which includes a number of *methods*. A method is typically a function or a procedure that performs a specific action and can be called by the software using the COM object (the *client* of that object). The methods that make up each interface are usually related to one another in some way. Clients can access the services provided by a COM object only by invoking the methods in the object's interfaces—they can't directly access any of the object's data.

For example, imagine a spell checker implemented as a COM object. This object might support an interface that includes methods such as LookUpWord, AddToDictionary, and RemoveFromDictionary. If the object's developer later wanted to add support for a thesaurus to this same COM object, the object would need to support another interface (perhaps with a single method such as ReturnSynonym). The methods in each interface collectively provide related services, either spell checking or access to a thesaurus.

The methods in each interface usually focus on supplying a particular service

Or imagine a COM object representing your bank account. It might support an interface that you access directly, one with methods such as Deposit, Withdrawal, and CheckBalance. This same object might support a second interface containing methods such as ChangeAccountNumber and CloseAccount, which can be invoked only by bank employees. Again, each interface contains methods that are related to one another.

---

1 Don't confuse COM objects with the objects in programming languages such as C++. Although they're similar in some ways, they're not the same. Later, this chapter describes how COM objects relate to other kinds of objects.

Figure 1-3 illustrates a COM object. Most COM objects support more than one interface, and the object in Figure 1-3 is no exception: it supports three interfaces, each represented by a small circle attached to the object. The object itself is always implemented inside a server, shown as the rectangle around the object. This server can be either a dynamic-link library (DLL), which is loaded as needed when an application is running, or a separate process of its own.

A COM object is implemented inside a server and usually supports multiple interfaces

A COM object's services are accessed via its interfaces.

Figure 1-3

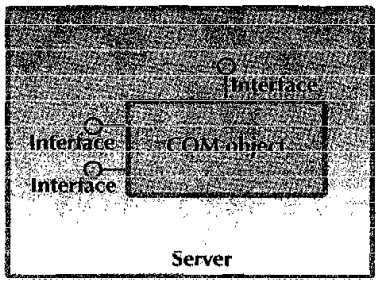
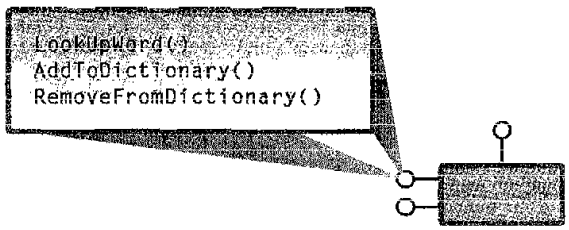


Figure 1-4 shows a close-up of a single interface supported by this COM object. This interface allows access to a spell checking service and contains the three methods previously listed. If another of the object's interfaces allowed access to the thesaurus service described earlier, a close-up of it would contain only the ReturnSynonym method. (In fact, this diagram is a bit simplified—all interfaces actually include a few more standard methods, which aren't shown here.)

Each interface provides one or more methods.

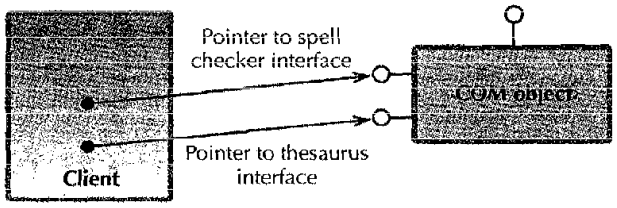
Figure 1-4



A client uses an interface pointer to invoke an interface's methods

To invoke the methods in a COM object's interface, a client must acquire a pointer to that interface. A COM object typically provides its services through several interfaces, and the client must have a separate pointer to each interface whose methods it plans to invoke. For example, a client of our sample COM object would need one interface pointer to invoke the methods in the object's spell checker interface and another pointer to invoke the method in the object's thesaurus interface. Figure 1-5 shows a client with pointers to two interfaces on a single COM object.

Figure 1-5 ***A client with pointers to two of a COM object's interfaces.***



Each COM object is an instance of a class

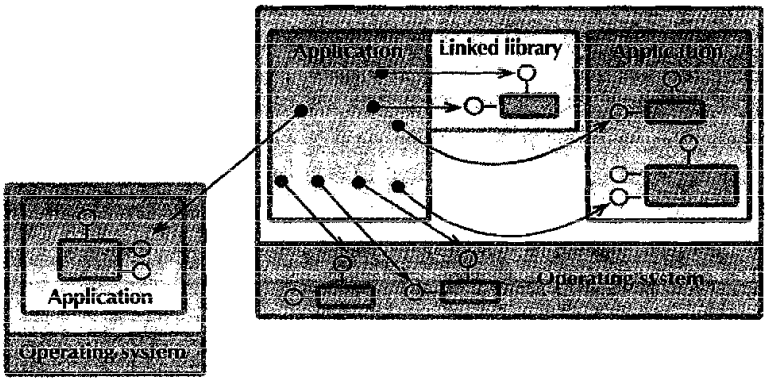
Every COM object is an instance of a specific *class*. One class, for example, might contain objects that provide spell checking and thesaurus services, while another might contain objects representing bank accounts. Typically, you must know an object's class to begin running an actual instance of that object, which you can do using the *COM library*. This library is present on every system that supports COM, and it has access to a directory of all available classes of COM objects on that system. A client can, for example, call a function in the COM library specifying the class of COM object it wants and the first supported interface to which it wants a pointer. (The COM library provides its services as ordinary function calls, not through methods in COM interfaces.) The COM library then causes a server that implements an object of that class to start running. The library also passes back to the initiating client a pointer to the requested interface on the newly instantiated COM object. The client can then ask the object directly for pointers to any other interfaces the object supports.

Once a client has a pointer to the desired interface on a running object, it can start using the object's services simply by invoking

the methods in the interface. To a programmer, invoking a method looks like invoking a local procedure or function. In fact, however, the code that gets executed might be running in a library or in a separate process or as part of the operating system or even on another system entirely. With COM, clients don't need to be aware of these distinctions—everything is accessed in the same way. As shown in Figure 1-6, one common model is used to access services provided by all kinds of software.

With COM, an application accesses an object's services (no matter where that object resides) by invoking a method in an interface.

Figure 1-6



COM and Object Orientation

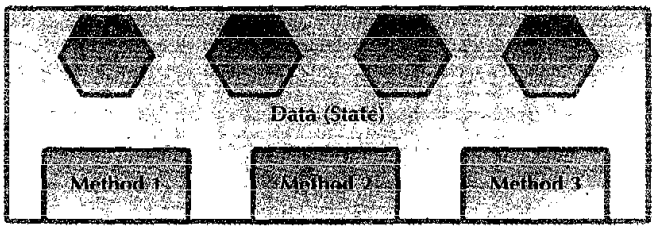
Objects are a central idea in COM. But how COM defines and uses objects sometimes differs from the way objects are used in other popular object technologies. To understand how COM relates to other object-oriented technologies, it's useful to describe what's commonly meant by the term *object-oriented* and then see how COM fits in.

An object is a combination of data and methods

Defining an object The term *object* has been blurred by marketeers trying to latch on to the latest fad, but in the minds of most, object-oriented technologies have a few key characteristics. Chief among these is a common notion of what constitutes an object. There is widespread agreement that an object consists of two elements: a defined set of data (also called *state* or *attributes*)

and a group of methods. These methods, commonly implemented as procedures or functions, allow a client of the object to ask the object to perform various tasks. Figure 1-7 shows a simple picture of an object.

Figure 1-7 ***An object has both methods and data.***



Unlike COM, most popular object technologies allow only a single interface per object

So far, so good—objects in COM are exactly like this. But in most object technologies, each object supports a single interface with a single set of methods. In contrast, COM objects can—and nearly always do—support more than one interface. An object in C++, for example, has only a single interface that includes all the object's methods. A COM object, with its multiple interfaces, might well be implemented using several C++ objects, one for each COM interface the object supports (although C++ isn't the only language that can be used to build COM objects).[2]

Another familiar idea in object technology is the notion of class. All objects representing bank accounts, for example, might be of the same class. Any particular bank account object, such as the one representing your account, is an *instance* of this class.

In COM, a class identifies a particular implementation of a set of interfaces

COM objects, too, have classes, as already described. In COM, a class identifies a specific implementation of a set of interfaces. Several different implementations of the same set of interfaces can exist, each of which is a different class. From the client's point of view, what matters are the interfaces. How those interfaces are implemented, which is what the class really indicates, isn't the

2 It's worth noting that, like COM objects, objects in the Java programming language can have multiple interfaces. In fact, as described in chapter 11, Java is a good fit for developing COM objects in several other ways, too.

client's concern. This ability to work identically with different kinds of objects, each supporting the same interfaces but implementing them differently, is called *polymorphism*. It's described a bit more in the next section.

Encapsulation, polymorphism, and inheritance If a technology models things as groups of methods and data and then organizes those groups into classes, is that sufficient to qualify it as object-oriented? Although there's plenty of debate, the answer from most quarters is no. In general, being object-oriented requires support for three more characteristics: encapsulation, polymorphism, and inheritance.

Encapsulation prevents a client from directly accessing an object's data

Encapsulation means that an object's data is not directly available to the object's clients. Instead, that data is encapsulated, hidden away from direct access. The only way to access the object's data is by using that object's methods. These methods collectively present a well-defined interface to the outside world, and it's only through this interface that a user of the object can read or modify its data. Encapsulation protects the object's data from inappropriate access and lets the object itself control how the data is accessed. By preventing inadvertent, incorrect changes from being made directly to an object's data, encapsulation can help enormously in the creation of better software.

COM objects support encapsulation

C++ provides direct support for encapsulation (although it also offers ways around it). If a programmer inappropriately attempts to directly modify an object's data, the compiler can flag the attempt as an error. Although COM isn't a programming language, the same idea holds. A client can access a COM object's data only through the methods in that object's interfaces. A COM object's data is encapsulated.

Polymorphism lets a client treat different objects as if they were the same

The second defining characteristic of object-oriented technologies is *polymorphism*. Simply put, polymorphism means that a client can treat different objects as if they were the same, and yet each object will behave appropriately. For example, think of an object representing your checking account. This object probably has a

Withdrawal method, which you implicitly call each time you write a check. You might also have an object representing your savings account, an object that also has a Withdrawal method. To a client, these two methods look just the same; and when either method is invoked, the same thing happens: the object's balance shrinks.

Different objects can implement the same method in different ways

In fact, however, the implementation of these two methods might be quite different. The implementation in the savings account object might simply check the requested debit amount against the account balance. If the debit amount is smaller than the balance, the request succeeds; if not, it fails. The Withdrawal method in the checking account object, on the other hand, might be a bit more complex. Checking accounts commonly offer an automatic loan up to a certain amount if a check would otherwise bounce. In implementing the Withdrawal method, the checking account object could check the requested debit amount against both the current account balance and the maximum loan currently available. In this case, the request succeeds and the check clears if the requested debit amount is less than the sum of the current balance and the available loan amount.

To a client, these two Withdrawal methods look alike; the differences in their implementation, important as they are, are hidden. This ability to treat different things as if they were the same, with each nevertheless behaving appropriately, is the essence of polymorphism. This example also demonstrates the great benefit of polymorphism: clients can remain blissfully unaware of differences that don't concern them, which simplifies the development of client software.

COM objects provide polymorphism

COM objects fully support this idea. It's entirely possible for two objects of different classes to present the same interfaces or perhaps only a single common method definition to their clients, even though each object implements the relevant methods differently.

The final defining characteristic of traditional object-oriented technologies is *inheritance*. The idea is simple: given an object,

you can create a new object that automatically includes some or all of the features of the existing object. Just as a man might, with no effort on his part, inherit male-pattern baldness from his parents, an object can automatically inherit characteristics of another object.

Inheritance allows a new object to build on an existing object

There are various kinds of inheritance. One distinction that's worth making here is between *implementation inheritance* and *interface inheritance*. With implementation inheritance, an object inherits code from its parent. When a client of the child object calls one of the child's inherited methods, the code of the parent's method is actually executed. With interface inheritance, however, the child inherits only the definitions of the parent's methods. When a client of the child object calls one of these methods, the child itself must provide the code for handling the requests.

Implementation inheritance and interface inheritance are different

Implementation inheritance is a mechanism for code reuse, one that's widely used in languages such as C++ and Smalltalk. Interface inheritance, in contrast, is really about reusing a specification—the definition of the methods that an object supports. An important reason for using interface inheritance is that it makes it easier to provide polymorphism. Defining a new interface by inheriting from an existing interface guarantees that an object supporting the new interface can be treated like an object that supports the old one.

Interface inheritance reuses a specification rather than actual code

Programming languages such as C++ support both implementation inheritance and interface inheritance. COM objects, however, support only interface inheritance. COM's creators believed that, given COM's very general applicability, supporting implementation inheritance was an inappropriate (and even potentially dangerous) way for one COM object to reuse another. For example, because implementation inheritance often exposes the inheriting object to details of its parent's implementation, it can break the encapsulation of the parent. Supporting only interface inheritance, as COM does, allows reuse of a key part of another object—its interface—while avoiding this problem.

COM objects support only interface inheritance

COM objects can reuse code through containment or aggregation

But without implementation inheritance, how can one COM object reuse another's code? In COM, this is done with mechanisms called *containment* and *aggregation*. With containment, one object simply calls another object as needed to help carry out its functions. With aggregation, an object presents one or more of another object's interfaces as its own; what a client sees as a single object providing a group of interfaces is in fact two or more objects aggregated together. As you might imagine, aggregation takes a bit more work to implement than containment does, but both provide an effective way to build on existing COM objects.

Is COM really object-oriented? COM has a great deal in common with other object-oriented technologies. Its basic notion of an object as a collection of data and methods resembles that idea in languages such as C++, although COM allows a single object to have multiple interfaces. COM also provides encapsulation, polymorphism, and interface inheritance, but it reuses code through containment and aggregation rather than through implementation inheritance. Objects are fundamental to COM, but the way those objects are defined and exactly how they behave differ somewhat from other widely used object-oriented technologies.

COM is object-oriented, but it differs from other popular object-oriented technologies

So is COM really object-oriented? The answer depends on what this question means. If it's asking "Are COM objects exactly like objects in languages such as C++?", the answer is obviously no. This shouldn't be too surprising, since COM solves a problem that is quite different from the one addressed by an object-oriented programming language. But if the real question is instead "Does COM provide the key features and benefits of objects?", the answer is just as obviously yes, and it's this second question that really matters. The goal isn't to get lost in debates about whose definitions to use. The goal is to write better software.

COM and Component Software

Hardware has progressed faster than software

In the past 35 years, hardware designers have gone from building room-size computers to creating lightweight laptops based on tiny, powerful microprocessors. In the same 35 years, software

developers have gone from writing large systems in assembler and COBOL to writing even larger systems in C and C++. While this is (arguably) progress, the software world isn't advancing at the same rate as the hardware world. Just what do hardware designers have that software developers don't?

Hardware design is aided by heavy reuse of existing components

The answer is components. If hardware engineers had to start from sand every time they built a new device, if their first step was always to extract the silicon to make a chip, they wouldn't progress very quickly, either. But, of course, this isn't what they do. Instead, a hardware designer typically builds a system out of prepackaged components, each of which performs a particular function, and each of which provides a defined set of services through well-specified interfaces. Hardware designers can greatly simplify their task by reusing the work of others.

Reuse is also a path to creating better software. Software developers today often start with something that's not too far from sand and then proceed to retrace the steps of a hundred programmers before them. The result is often very good, but it could be even better. Creating new applications from existing, tested components is likely to produce more reliable code. And, just as important, it can be much faster and significantly cheaper.

Component software applies this idea to the creation of new software

This idea of defining reusable parts, each presenting its services through well-specified interfaces, is exactly the approach that COM takes. COM objects provide an effective mechanism for software reuse by allowing the creation of discrete, reusable components. These components can act much like the various chips that hardware designers use, with each one supporting a specific function. Perhaps because of this analogy, this approach has become known as *component software*.

Existing approaches to software reuse haven't been sufficient

This is hardly a new idea. Developers have recognized the potential power of software reuse since the days before compilers. Some of the strictures on reuse are cultural—incentives in many organizations encourage reinvention rather than reuse, for example. But technology also constrains the potential for reuse.

Existing reuse mechanisms, important as they are, don't go far enough. To understand why this is so, it's helpful to examine the two reuse schemes that are most commonly seen today: libraries and objects.

Software reuse through libraries can help

As a mechanism for reuse, libraries have a lot to offer. This is especially true of dynamic-link libraries, which can be loaded on demand and are typically shared rather than statically linked into only one application. Libraries are familiar and easy to use. Since they can be distributed in binary form, there's no risk of revealing proprietary source code to prying eyes. And, with a little care, a program written in one language can call the routines from a library written in a different language. Libraries aren't without problems, however. One significant headache is the difficulty of adding functionality: how can you install a new version of a library without breaking applications that use the old version? And how can you easily and safely have more than one implementation of the same library on your system, which might be required in some circumstances? Libraries just aren't enough.

Software reuse with objects can also help

By encapsulating data and methods, objects can also provide a clean way to package reusable chunks of functionality. Much like traditional libraries, objects that solve specific problems can be created once and reused many times. But objects have even more to offer than libraries do. Through inheritance, one object can reuse another object's interface definition or its code or both. And polymorphism simplifies reuse by hiding irrelevant differences from an object's clients.

But no large market in reusable objects exists today

Despite these advantages, object technology hasn't achieved its full potential for enabling software reuse. To see why, consider this: why can't an organization that wants to write a new application start the process by visiting the software store, checking a catalog, or searching the World Wide Web for the objects it will need? Why is there no large market in business-focused, reusable objects? Hardware developers benefit from this kind of market, so why can't creators of software have one, too? Why is there no object bazaar, rich with choices?

The answers are rooted in the object technologies we use today. Object-oriented languages such as C++ were designed to allow reuse within workgroups or, at most, a single organization. While you can certainly find some reusable C++ objects for sale, the kind of worldwide object bazaar envisioned here isn't feasible with existing technology. Standing in the way are three major problems.

Traditional object technologies present three obstacles to creating a component software market

The first and perhaps most important problem is that standards for linking binary objects together don't really exist. Although you can compile a C++ object and then use that compiled binary object from a library, this is guaranteed to work only when the same compiler is used for both the library and the application using the library. C++ doesn't have cross-compiler standards for the format of binary objects, so building and distributing binary object libraries is problematic, at best. As a result, currently available C++ object libraries almost always include source code. A related point: reusing code through implementation inheritance tends to bind parent and child objects together tightly. The creator of the child object should usually have access to the parent's source code, if only to know exactly what happens when an inherited method is called.

Problem 1: Distributing objects with their source code

Is it reasonable to expect that the creators of the software available in our hypothetical object bazaar will be willing to distribute their source code, thus revealing their proprietary secrets? The answer appears to be no, since no such bazaar exists. Although source-code–based reuse is entirely reasonable within a development group or even inside a single company, for a worldwide object bazaar binary distribution is essential.

The second problem is that, despite its dominance in object-oriented development, C++ is not the only language in the world. An object written in C++ can't be easily reused in, say, a Smalltalk program. And what about tools such as Powersoft's PowerBuilder or Microsoft's Visual Basic? While one can argue about whether these environments are really object-oriented, one cannot argue with their popularity. An object bazaar should offer objects that

Problem 2: Reusing objects across different languages

can be used and reused across various languages and development environments, but currently it's difficult to reuse an object written in one language in an application written in another language.

Problem 3: Relinking or recompiling an entire application when one object changes

The third problem is this: if you create an application out of objects written in a language such as C++ and then decide to change one of the objects, you must at best relink, and perhaps even recompile, the application. If several applications on one system use this changed object, you must relink or recompile all of them. Ideally, you'd have a way to drop in a new version of a single object and have all applications that use this object automatically use the new version. And, of course, this should happen without relinking or recompiling any of those applications.

COM solves all three problems

All of these problems are solved by COM. COM objects can be packaged into libraries or executable files and then distributed in a binary format (without the source code). Since COM defines a standard way to access these binary objects, COM objects can be written in one language and used in another. And since COM objects are instantiated as needed, when a new version is installed on a system, all clients will automatically get the new version the next time they use the object. COM offers the reuse benefits of both libraries and objects, along with other benefits that neither libraries nor objects alone can provide, chief among them a common approach to accessing all kinds of software services.

COM aims to create a large market in reusable components

COM brings the benefits of widespread reuse, prevalent for so long in hardware design, to the creation of software. In fact, sites full of COM-based components already exist on the World Wide Web, where you can browse or even download components, and magazines are chock-full of component advertisements. The object bazaar is becoming a reality, allowing software developers to create applications that are at least partially built from reusable parts. COM's general service architecture is useful for many tasks, but supporting the creation of component software was perhaps the single most important goal in the minds of its creators.

The Benefits of COM

Anything that simplifies the complex endeavor of creating large pieces of software is good. The conventions defined by COM do this in several ways.

COM offers the benefits of object orientation

COM offers a useful way to structure the services provided by a piece of software. Developers can design their implementation by first organizing it into COM objects and then defining the interfaces for each object. This is one of the traditional benefits of an object-based approach to design and development. And, as just described, COM goes further by allowing developers to create software components that can be safely distributed and reused in a variety of ways.

COM provides consistency

A second benefit of COM has already been mentioned: consistency. By providing a single approach for accessing all kinds of software services, COM simplifies the problems developers face. Whether the software in question is in a library, in another process, or part of the system software, you can always access it in the same way. A side effect of this consistency is that COM tends to blur the distinction between applications and system software. If you can access everything as a COM object, you'll perceive little significant difference between these two kinds of software, which have traditionally been quite distinct. Instead, you can develop applications that build on the software services available in your environment, whatever they happen to be and whoever happens to provide them.

COM is language independent

In addition, COM is blind to the programming language being used. Because COM defines a binary interface that objects must support, you can write COM objects with any language that can support this interface. You can then use any language capable of making calls through this binary interface to invoke the methods in the interfaces of those objects. An object and its client neither know nor care what language the other is written in. While it's fair to say that some languages are better suited for use with COM than others, COM itself strives to be language independent.

COM's approach to versioning is simple and efficient

Another benefit of COM stems from its approach to one of the most persistent problems in developing and deploying software: *versioning*—that is, replacing an existing version of software with a new version that offers new features, while not breaking any existing clients. COM objects provide a simple answer, based on an object's ability to support more than one interface. As explained earlier, a COM object's client must acquire a pointer to each specific interface it needs to use. To add features in a new version of a COM object, then, you can simply offer the new features through a new interface on the object. Existing interfaces aren't changed (in fact, COM prohibits changes to existing interfaces), so clients using those interfaces are unaffected. And these existing clients never ask for pointers to the new interfaces. Only new clients know enough to ask for the interfaces that offer the new features, and so only new clients are affected by the new version.

COM also solves the other side of the versioning problem: what if a client expects an object to provide certain functionality, but the object hasn't yet been updated to offer it? The client requests a pointer to the interface through which this service would be available but gets nothing in return. Because COM supplies a clean way to learn that an object isn't all the client hoped it would be, developers can write clients to handle this situation gracefully instead of crashing. This simple, clean approach to versioning, which allows independent updates to both clients and the objects they use, is among COM's biggest contributions.

COM is used widely throughout Microsoft's product line

Microsoft itself is adopting COM in most of its products. The company is using COM to define extensions to Microsoft Windows and Microsoft Windows NT, applying it in various ways in many Microsoft applications, and using it to define standard interfaces for many kinds of services. COM's approach can be applied profitably to the development of all kinds of software.

COM's Availability

COM, which was developed by Microsoft, was originally made available on Windows and Windows NT. Microsoft now also provides support for COM on the Macintosh. Although Microsoft

does not support COM on other operating systems, this void has been filled by third parties. Several companies, large and small, provide implementations of COM and various COM-based technologies on a wide range of operating systems. Software developed using COM objects will be available on all kinds of systems, ranging from workstations that run Windows and Windows NT to IBM mainframes that run MVS. And, as you'll see later, Distributed COM (DCOM) allows COM objects on all kinds of systems to interact. COM's increasingly central role in software developed for Windows and Windows NT, coupled with the ubiquity of these systems, suggests that this new approach to creating software will work its way into all parts of the enterprise.

COM will be available on many operating systems

Defining Standard Interfaces with COM

COM provides the basic mechanisms needed for one chunk of software to provide services to another through well-defined interfaces implemented by COM objects. But who defines those interfaces? Unless a COM object and its client agree on what interfaces exist, what methods those interfaces contain, and what the methods actually do, it's not possible for them to accomplish anything useful.

In some cases, developers must define application-specific interfaces. For example, an investment bank creating its own custom software for carrying out trades might decide to design and build that software using COM. The software's developers can define appropriate custom interfaces as they see fit and then implement support for those interfaces in their own COM objects. There's no need to contact or seek approval from Microsoft.

Application developers can define interfaces as they see fit

But suppose that all investment banks have similar requirements for objects and their interfaces. Why not bring them together to define industry-standard interfaces for these objects? This would allow the creation of a market for standard components produced by competing companies. OLE Industry Solutions, a Microsoft-sponsored program to define these sorts of interfaces, has precisely this goal. Through this program, groups from financial

The OLE Industry Solutions program is designed to create industry-standard interfaces

companies, healthcare organizations, providers of point-of-sale equipment, and others have defined standard interfaces for components useful in each area.

There are other kinds of services where new standard interfaces might become even more well known. For example, suppose that the owners of an operating system decided to make the services of its file system available via COM. They would need to define one or more COM objects, each with a specific class and supporting a defined set of interfaces. Then they would have to make those interface definitions available to the potential users of the COM objects—that is, to developers of applications that use the new file system.

Microsoft itself defines standard interfaces in many cases

The original problem addressed by OLE, creating compound documents, is another example of the need for standard interfaces. A compound document (as you saw in Figure 1-1 on page 2) contains elements from two (or possibly more) applications that share a single window on the user's screen, allowing the user to work with information from both applications. Clearly, both applications must cooperate to make this possible, providing services to each other that allow them to present a seamless interface to the user. They can do this by each supporting certain COM objects, each of which in turn supports specific interfaces. And since the goal is to allow all kinds of applications to cooperate in a standard way, someone must define and publicize the required COM objects and interfaces.

Every ActiveX and OLE technology defines a set of interfaces using COM

Defining (and sometimes implementing) standard interfaces to perform well-defined functions is what ActiveX and OLE are all about. In Structured Storage, for example, a COM-based technology provided with Windows and Windows NT, COM objects and interfaces define elements of a file system. The technology for creating compound documents, one of the most commonly supported COM-based technologies, is implemented by COM objects with standard interfaces that allow applications to share screen real estate and create compound documents.

ActiveX and OLE technologies are nothing more than software that provides services to clients through COM interfaces supported by COM objects. Various parts of ActiveX and OLE define standard interfaces for various purposes. Some of those interfaces are supported by system software, as in the file system example; others, like those for creating compound documents, must be supported by individual applications. In either case, the fundamental mechanism used to provide services to clients of the software is the same: COM.

Describing ActiveX and OLE Technologies

OLE, which once again refers only to technologies for creating compound documents, and the broad set of technologies assigned the ActiveX label are all built using COM. Many of these technologies have their roots in compound documents, but others address entirely different problems. This section provides a brief introduction to the most important COM-based technologies.

Automation

Automation provides programmability

Spreadsheet applications, word processors, and other personal productivity software give people all sorts of useful capabilities. Why not let other software access those capabilities, too? For this to be possible, applications must expose their services to programs as well as to people. In other words, they must be programmable. Providing this programmability is the goal of *Automation* (originally known as *OLE Automation*).

Automation clients typically access an object's methods via a dispinterface

An application can become programmable by exposing its services through ordinary COM interfaces. This is seldom done, however. Instead, applications expose their services through *dispinterfaces*. A dispinterface is much like the interfaces described so far—it has methods, clients access those methods using an interface pointer, and so on—but it also differs in significant ways. In particular, dispinterface methods are much easier to invoke from clients written in simple languages such as Visual

Basic. This is a crucial point because Visual Basic and tools like it are the first choice for most people who want to write programs that access an application's internal services.

To get a sense of how useful this idea can be, think of Microsoft Excel. This spreadsheet program offers a wide range of functions that are typically accessed directly by the person using Excel. It's also possible, of course, to create complete applications using Excel by writing them in Excel's built-in macro language.

Excel allows access to its services through dispinterfaces

Today, however, Microsoft Excel supports Automation—that is, Excel makes its internal services available through dispinterfaces supported by various COM objects, which provide methods such as Average, CheckSpelling, and many more. Applications built on Excel no longer are restricted to using Excel's built-in macro language but instead can be written in virtually anything. Excel itself is no longer only a tool for end users—it's now a toolbox for application builders, too.

Many other applications also support Automation

This same feature, programmatic access to internal services through Automation, is supported by a host of other applications. This ability to easily access the powerful features offered by an existing application is what makes Automation among the most widely used COM-based technologies. For a more detailed discussion of Automation, see Chapter 4.

Persistence

COM objects can make their data persistent

Objects have data and methods, and many objects need a way to store their data when they're not running. In the jargon of the cognoscenti, an object needs a way to make its data *persistent*, which typically means storing that data on disk. COM objects have many choices for how to accomplish this. One of the most commonly used is known as *Structured Storage*.

To understand Structured Storage, think first about how applications save their data in ordinary files. Traditional file systems allow applications to share a single disk drive without getting in one

another's way. Each has its own files and maybe even its own directories to work with, independent of what other applications might be doing. Applications don't need to cooperate in storing their data, since each one can be assigned its own storage area.

With COM, however, the situation gets more complicated. Because COM allows all kinds of software to work together using a single model, independently developed COM objects might become part of what the user sees as a single application but might still need to store their data on disk separately. While each COM object could use its own file, to the application's users the objects are invisible—this is a single application—and having to keep track of multiple files is unlikely to make users very happy.

Structured Storage allows COM objects to share a single disk file

What's needed is a way for multiple COM objects to share a single file. This is exactly what Structured Storage provides. By essentially building a file system inside each file, Structured Storage allows the components comprising a single application to each have its own discrete chunk of storage space, its own "files." To the user, only a single file exists. To the application, however, each component has a private area for storing data, all of which are actually contained within a single disk file.

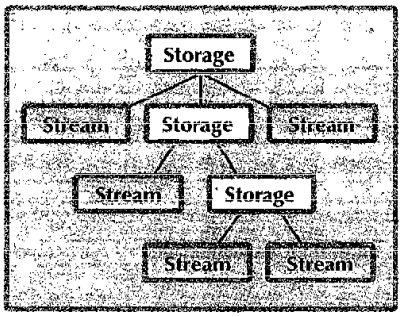
Structured Storage organizes a file into storages and streams

To make this work, Structured Storage defines two kinds of COM objects, each supporting appropriate interfaces. Called *storages* and *streams* (illustrated in Figure 1-8 on the following page), these objects are analogous to the directories and files, respectively, of common file systems. With Structured Storage, a single file can contain data stored by many COM objects, each storing its data in its own storage or stream. Just as a conventional file system allows different applications to share a single disk drive, Structured Storage provides a way for different applications to share a single file.

There's more to persistence than Structured Storage, however. A COM object can save its persistent data in other ways, such as in an ordinary file or even on the World Wide Web. Also, an object

must supply a way for its clients to tell it when to load and save its persistent data. To allow this, an object can support one (or perhaps more) of several standard interfaces defined for this purpose. Chapter 5 presents a more complete description of persistence in COM objects.

Figure 1-8 ***With Structured Storage, a single file contains several storages and streams.***



Monikers

A client can create and initialize a COM object

Imagine an instance of a COM object that represents your bank account. To access your account, a client needs to start this object and then have the object load its data (your account balance and other information). COM offers a way to do this—that is, it provides mechanisms that allow a client to instantiate and initialize a COM object.

To perform this task, the client needs to know quite a bit. It must know, for example, how to locate the correct data for your account and how to tell the appropriate COM object to load this data. While it's sometimes reasonable to expect the client to know all this, it would be nice if there were some way to hide this detail, to let the client simply say, "Create this object instance" (your bank account) and have everything happen automatically.

A moniker knows how to create and initialize another object

This is exactly what monikers do. A *moniker* is itself a COM object, but it has a very well-defined purpose: each moniker knows how to create and initialize one other object instance. If I

had a moniker for your bank account object, for instance, I could ask that moniker to create, initialize, and connect me to your account. All the details of what's required to do this are hidden from me (the client). If I wanted to work with two bank accounts, using monikers to access them, I'd need two separate monikers, one for each account object. In general, monikers aren't required in the COM environment; they just make things easier for the client. Monikers are described in Chapter 6.

Uniform Data Transfer and Connectable Objects

Uniform Data Transfer lets all kinds of software exchange data in a common way

Exchanging data is a fundamental software operation. Applications copy data back and forth, for example, when their user moves data via the clipboard. Various kinds of system software, such as device drivers, provide information from their devices to software using those devices. Given the plethora of reasons for different chunks of software to exchange data, it's not surprising that an overabundance of schemes have been invented to do it.

In the COM world, *Uniform Data Transfer* is the standard way to exchange information. As with all ActiveX and OLE technologies, the applications involved must support particular COM interfaces. The methods in these interfaces define standard ways to describe the data being moved, to specify where that data resides, and to actually move it. They even define a simple mechanism that lets one application inform another when an interesting piece of data becomes available. Although it's hardly the most exciting thing that COM has to offer, Uniform Data Transfer plays an important part in much of the work that COM-based applications perform.

While it's useful in some situations, the simple scheme defined by Uniform Data Transfer for notifying a client when interesting data has appeared isn't entirely sufficient, however. A COM-based technology known as *Connectable Objects* has been created to address this deficiency. By providing a more general mechanism through which an object can talk back to its client, Connectable Objects makes it easy for clients to receive notifications of interesting events. Both Uniform Data Transfer and Connectable Objects are discussed in Chapter 7.

Compound Documents

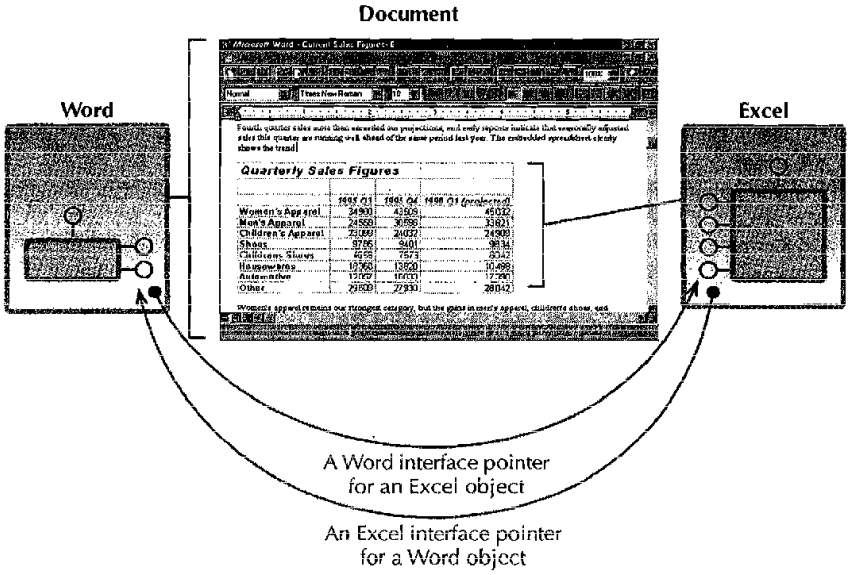
Applications get more complicated every day. Word processors add graphical capabilities, spreadsheets add charting functions, and it can seem as if we'll eventually wind up using one big application for everything. But that isn't really the aim; rather, the goal is integration among different applications. A word processor doesn't need to add graphing functions, for instance, if you can use an existing graphing application from within the word processor. The intent is to have applications work together smoothly. A user should be able to see what appears to be a single document but have different applications cooperate to work on various pieces of that document.

OLE technology allows the creation of compound documents

The OLE technology (formerly known as *OLE Documents*) addresses this problem. By supporting appropriate COM objects, each with its own set of interfaces, separate applications can cooperate to present one compound document to the user, as shown in Figure 1-9. These interfaces are completely generic—neither application knows what the other one is. A user might, for example, work with a Word document that contains an Excel spreadsheet, as shown in the figure. When the user modifies the text, Word is in control. Double-clicking on the spreadsheet part of the document silently starts Excel, allowing the user to manipulate the spreadsheet's data using Excel. The word processor doesn't need to build in the functions of a spreadsheet; with OLE, an existing spreadsheet application can simply be plugged in as needed.

The standard interfaces defined by OLE enable this kind of interaction among all sorts of applications from any vendor, not just spreadsheets and word processors produced by Microsoft. You can include sound in graphs, create presentations with integrated video clips, and more. Many applications today, from a wide range of vendors, support OLE as a way to interact with other applications.

Documents can contain elements managed by separate applications. ***Figure 1-9***



When you create a compound document with OLE, one application always acts as the *container*. As the name implies, a container defines the outermost document, the one that contains everything else. In Figure 1-9, Word is the container. Other applications, called *servers*, can place their documents within the container's document. In Figure 1-9, for example, Excel is acting as a server.

Applications act as containers and servers

Using OLE, a server's document can be *linked* to or *embedded* in the container's document. If the server's document is linked, it's stored in a separate file, and only a link to that file is stored in the container's document. (The link is actually a moniker.) If a server's document is embedded, that document is stored in the same file as the container's document. (The two applications share a single file using Structured Storage.)

Documents can be linked to or embedded in other documents

Creating compound documents was the problem that led to the creation of COM. Although COM is used much more widely today, the fingerprints of compound documents are visible on many

COM-based technologies. This challenging problem motivated the design of a large number of core technologies in this area. Chapter 8 describes the interfaces that OLE containers and servers must support and explains how those interfaces work to give a user the illusion of a single document.

ActiveX Controls

If you want to include a spreadsheet in a text document, why should you be forced to use all of Excel? If you need only basic spreadsheet functions, maybe you can get by with a simpler, faster, and probably cheaper spreadsheet component. Or suppose that you're using Visual Basic to build an application that needs to include some spreadsheet functionality. It'd be great to just plug in the basic functions you need without dragging along (or paying for) a complete spreadsheet application. In fact, you might like to build your entire application largely from existing components that you plug together.

ActiveX Controls defines standard interfaces for reusable components

This desire is what led to the idea of component software, an area where COM has much to contribute. You can build reusable components solely with COM itself, but it's also useful to define some standard interfaces and conventions for this purpose. Using these, you can build components that perform common tasks, such as providing a user interface and sending events to a client, in a common way. The *ActiveX Controls* specification defines these standards.

An ActiveX control is a stand-alone software component that does specific things in a standard way. Developers can plug one or more ActiveX controls into an application created in, say, Visual Basic to take advantage of existing software functionality. The result is software built largely from prefabricated parts—that is, component software.

ActiveX controls were originally called OLE controls or OCXs

ActiveX controls were originally known as OLE controls or OCXs. Microsoft changed the name to reflect several newly defined features that make these controls much more usable with the

Internet and the World Wide Web. For example, an ActiveX control can store its data on a page somewhere on the Web, or it can be downloaded from a web server and then executed on a client machine. And the container in which the control executes need not be a programming environment—it can instead be a web browser.

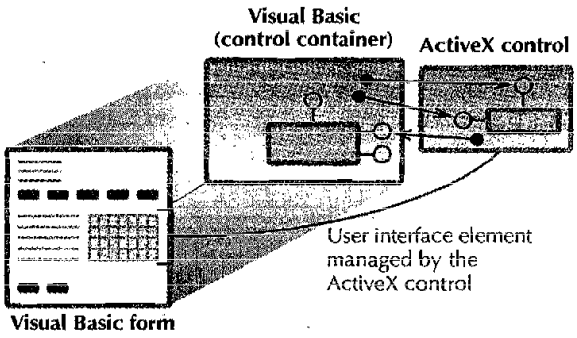
Hundreds of controls are available from dozens of companies, including spreadsheet controls, controls for mainframe data access, and many more. You can even select ActiveX controls by browsing sites on the World Wide Web and then download them for immediate use. By far the largest number of components today are built as ActiveX controls.

ActiveX controls rely on many other COM-based technologies

ActiveX controls are not separate applications. Instead, as shown in Figure 1-10, they're servers that plug into a control container. As always, the interactions between a control and its container are specified through various interfaces supported by COM objects. ActiveX controls actually make use of many other OLE and ActiveX technologies. Controls typically support the interfaces defined for embedding, for example, and they also commonly allow access to their methods via the dispinterfaces defined for Automation. ActiveX controls are described in Chapter 9.

Figure 1-10

The functions packaged in an ActiveX control can be used by any control container, such as Visual Basic or a web browser.

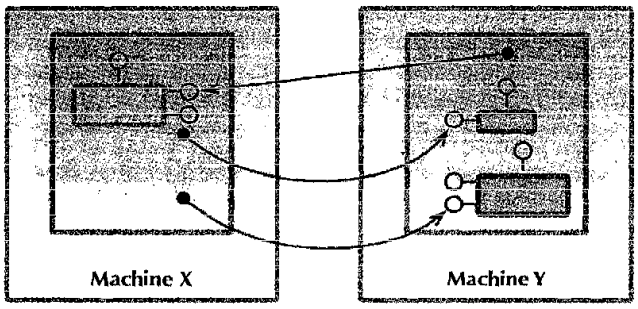


Distributed COM

Distributed COM allows clients to access COM objects on other machines

Although designed from the start to support distribution, the original implementation of COM ran on only a single system. COM objects could be implemented in DLLs or in separate processes running on the same machine as their client, but they couldn't reside on other machines in the network. *Distributed COM* (DCOM) changes this. With DCOM, COM objects can provide their services across machine boundaries, as shown in Figure 1-11.

Figure 1-11 ***Illustrating Distributed COM.***



DCOM uses RPC and supports security services

To achieve this, DCOM relies on remote procedure call (RPC). With RPC, a client can make what appears to be a local call to a component, although that call actually executes in an object across the network. DCOM also includes support for security services (controlling which clients can use which COM objects) and a way to specify the machine on which an object should be created. The services supplied by DCOM can be used to build secure, distributed, COM-based applications, and they are described in more detail in Chapter 10.

COM-Based Service Interfaces

It's often useful to have a common interface to access different implementations of a service. For example, the Open Database Connectivity (ODBC) interface built into Windows and Windows NT defines a group of C function calls that can be used to access any relational database management system. With the arrival of

COM, these kinds of interfaces can be specified in a common, object-oriented way. Microsoft has defined several such interfaces, including those for databases, transactions, and directory services based on COM.

DBMS services can be accessed using COM objects

Databases A database management system (DBMS) provides a way to organize, store, and retrieve information. DBMSs are widely used tools that underlie many applications. Local access to a DBMS is usually through a library linked into a client process or perhaps through some kind of interprocess communication. Really, though, a DBMS is simply a collection of services provided by one chunk of software to another. Why not model and deliver those services as COM objects?

A typical DBMS includes a query processor, various data storage mechanisms, and more. If standard objects and interfaces were defined and widely supported, a client could access various DBMSs in the same way or even use only the best parts from different ones. For instance, an application might benefit from using a data storage scheme from one DBMS and the query processor from another. And there's no reason why those same interfaces couldn't be applied more generally and used to access data that's not in a DBMS. Why not have a common approach to accessing relational data and, say, data stored in spreadsheets?

A COM-based database technology provides a way to access data stored in various ways

COM-based database technology (originally called *OLE Database*, or OLE DB) addresses these issues. By defining standard COM objects and interfaces for data access, this technology establishes a common means for clients to access data stored in various fashions. In many ways a generalization of ODBC, OLE Database goes beyond this earlier standard interface by viewing everything as COM objects. A source of data can be modeled as a DataSource object, for example, and then have a Command object defined for it. This Command object might specify an SQL query or another kind of command that manipulates the data. Every Command object provides an interface containing an Execute method, which (not surprisingly) executes the command. The result

is yet another object, called a Rowset, that contains the result of the command's execution. This object, in turn, supports an interface with methods that allow examination of the data contained in that object.

All of these objects are defined using COM, and all present their services through methods in COM interfaces. The result is an abstracted view of data access, one that can be implemented in numerous ways and for a range of data access mechanisms.

A transaction's operations either all succeed or all fail

Transactions In accessing data, especially distributed data, the notion of a transaction can be useful. Suppose that you'd like to modify two databases, but either both changes must happen or neither should—partial success is not acceptable. For example, to transfer $100 from your savings account to your checking account, two actions must occur: $100 must be subtracted from your savings account, and that same amount must be added to your checking account. If only the first request succeeds, you won't be happy. If only the second succeeds, the bank won't be happy (although you might be). To arrive at a consistent result, either both operations must succeed or both must fail.

To carry out this kind of indivisible atomic operation, you must define a transaction that includes both modifications. This service can be built into the data access mechanism itself, but a separate transaction service that can be used with different data access mechanisms is often a better idea. Once again, the goal is for one piece of software, the transaction service, to provide services to another. Why not describe this interaction using COM?

COM-based transactions technology models a transaction service as COM objects

Just as COM-based database technology models data access mechanisms using COM objects, COM-based transactions model a transaction service as COM objects. The objects defined include resource managers (for example, a DBMS), transaction coordinators, and the transactions themselves. And since transactions are common in data access, the interfaces defined for transactions are designed to work well with those defined for databases.

Directory services Much like a telephone directory, a directory service in a distributed environment allows its user to look up information.[3] With a telephone directory, you can find someone's phone number if you know that person's name. With a directory service, the client supplies a name, and the directory service returns information about the named item. For instance, a client might supply the name of a particular machine and get back the information it needs to contact that machine, such as a network address. Or a client might provide the name of a user and receive that user's e-mail address.

A directory service maps a name to information about the named object

A directory service is extremely useful in a distributed environment. Because no single directory meets everyone's needs, numerous directory services exist, and many different technologies are used. The most well known services include the Windows NT directory service, the internationally standardized but not widely used X.500, and the Novell Directory Service (NDS) used primarily with Novell NetWare, but there are many more.

COM-based directory services (originally known as OLE Directory Services or OLE DS) do for directory services what OLE Database does for database systems: they provide a common interface that can be used to access all kinds of directory services. Just as COM-based databases make it easier to create clients that must handle all kinds of data, COM-based directory services make it easier to create clients that must work with all directory services.

COM can be used to define a common interface to diverse directory services

To define this standard interface, the technology must provide a general way to model the information stored in diverse directories. Fortunately, directory services typically organize their information in some type of hierarchy. For example, all the information a company maintains in its directory might appear below a single

3 Don't confuse a directory service with a directory in a file system. The use of the word *directory* is broadly similar, but the two are not the same thing.

node that represents the company itself. The next level in the tree might contain entries for divisions of the company, and so on. Alternatively, an organization's directory hierarchy might reflect physical rather than organizational boundaries. One branch might contain entries for all the company's machines, for instance, while another might include entries for all the printers.

Directory entries are modeled as container objects or leaf objects

The COM-based solution is to model each directory entry as a COM object. Mirroring the kinds of objects in a hierarchy, every directory entry is either a *container object* or a *leaf object*. Regardless of the particular directory service being used, a client sees all the directory's entries as container objects or leaf objects. Container objects, as their name suggests, can contain leaf objects or other container objects. For example, a container object might represent a directory entry that is the parent node for all entries about printers. Below this container object might appear many different printer entries, each describing a specific printer. Each kind of object provides appropriate interfaces that let clients access the data and methods that object provides. The goal is to make life simpler for developers who create clients that use multiple directory services.

COM and Internet Technologies

Most of Microsoft's Internet-related technologies use COM

The Internet and the style of data access provided by the World Wide Web have crashed like a tidal wave on the shores of computing. Although Microsoft wasn't the first to recognize the impact this wave would have, the company wasted no time in responding once that recognition hit. Not surprisingly, most of the new technologies Microsoft has created in this area are built using COM. As described earlier, the ActiveX brand name originated in COM's collision with the Internet, although it has now spread to include many other COM-based technologies.

COM's component-oriented approach is applied to Microsoft's Internet and web technologies in several ways. For example, Microsoft's web browser, Internet Explorer, relies heavily on an

extension of OLE compound documents called ActiveX Documents. With this enhancement, a user can browse through many types of information in addition to the conventional Hypertext Markup Language (HTML) pages. The ActiveX Controls technology has been enhanced to allow a control's code and data to be intelligently downloaded as needed from a web server and executed inside a web browser. ActiveX Scripting provides a generic way for clients to execute scripts written in any scripting language, while the ActiveX Hyperlinks technology, based on monikers, allows the creation of Web-style hyperlinks not only between HTML pages but between all kinds of documents. All of these technologies are described in Chapter 11.

The Future of COM

The use of COM will continue to grow

From its humble beginnings as a way to create compound documents, COM has evolved into a fundamental underpinning for application and system software. COM has been so widely applied because the architecture it defines for providing software services offers an attractive solution to so many problems. Given this generality and its obvious benefits, the applications of COM described here are in all likelihood only the beginning. While the broad label applied to COM-based technologies has changed over time—from OLE to ActiveX—this matters little from a purely technical perspective. Whatever the name, the benefits of COM and applications of COM continue to spread throughout this part of the software world.

users from casually copying licensed components. While the mechanisms described here probably won't stop a determined pirate, they can prevent a significant amount of casual copying.

Extensions to ActiveX Controls

The Controls 96 specification extends the current definition of an ActiveX control

There's always room for improvement. A set of improvements to the ActiveX Controls specification, collectively referred to as *Controls 96*, defines a number of compatible extensions to the basics described so far. Those extensions include the following:

- Capabilities that allow a control's user interface to be of any arbitrary shape, not only a rectangle
- A new, faster initialization scheme that allows a control and its container to acquire all the initial interface pointers they need from each other through a single exchange
- Enhancements that allow a control to draw its user interface more efficiently and with less on-screen flicker

New features have also been added to make ActiveX controls better suited for the Internet

Another category of extensions to ActiveX controls grows out of the changes wrought by the Internet. As mentioned earlier, the once-onerous requirements for controls have been greatly relaxed, making it easier to create controls that can be swiftly downloaded across a slow Internet connection. To be truly useful in the Internet environment, however, controls also need a way to become active quickly while still downloading their persistent data in the background across a slow connection. And since this data might arrive in pieces, the control also must be able to notify its container that all the data has arrived. As Chapter 11 details, these features and more have been defined to allow the creation of Internet-aware controls. As support for these new features begins to appear in controls and control containers, the potential applications of ActiveX controls will become even broader.

Chapter Eleven

ActiveX, the Internet, and the World Wide Web

From its modest beginnings as a U.S. government–sponsored research network, the Internet has developed into a genuine phenomenon. By providing a global network linking millions of computers, the Internet makes possible things that once weren't even conceivable. And as ever-increasing numbers of homes and offices set up high-speed connections to this network, we can expect still more advances that are today inconceivable. The availability of cheap bandwidth—and the ubiquitous global network it makes possible—might prove to be a technical innovation as transforming as the invention of the microprocessor.

The growth of the Internet was driven largely by the World Wide Web

Like most new hardware-oriented innovations, the Internet expanded so rapidly because of a "killer" application, attractive enough to motivate people to use it. That killer app was the World Wide Web. The Web today is a major source of information and commerce for millions of people around the world. Web technology has found a receptive home in the business world, too, as corporate intranets based on Internet technologies have proliferated rapidly. Using these technologies, private organizations can build their own internal webs, allowing them to share information inside

an organization just as the Internet-based Web does externally. With its easy-to-use, easy-to-understand user interface, web technology has broad appeal.

COM is used throughout Microsoft's Internet and Web-related technologies

COM is fundamentally about defining the boundaries between pieces of software. The Internet has a major impact on those boundaries in several ways. The Web's browsing metaphor also affects how applications interact both with data and with their users, two more traditional concerns for technologies built using COM. To address these changes, several new COM-based technologies have been created, and others have been adapted for this new environment. This chapter explores these new and adapted technologies.[1]

ActiveX Documents

Embedded OLE documents, useful as they are, have some limitations

The conventions defined by OLE allow a user to edit an embedded document in place, much as if it were opened in a separate application. With an embedded Microsoft Excel spreadsheet like the one shown in earlier chapters, for instance, the user can activate the embedded object and have access to Excel's commands. Useful as this is, however, an ordinary embedded document doesn't suffice in every situation. Typically, for example, an in-place active document is relegated to whatever area on the screen its container is willing to allot, an area that's usually fairly small. In some cases, the user might want to have the embedded document completely take over the editing area of the user interface. Similarly, when a user prints an ordinary compound document, only the cached presentation appears for any embedded elements—the embedding server's own print functions can't be used. Having a way to access these functions and a few other extra features would let the user see the full functionality of the embedded application rather than just the (admittedly quite large) subset provided by OLE embedding and in-place activation.

1 An important note: this discussion is based on a pre-release version of the ActiveX Software Development Kit (SDK). It's possible that some parts will change before these technologies are finalized. Be aware that what's described here might not exactly match what is finally delivered.

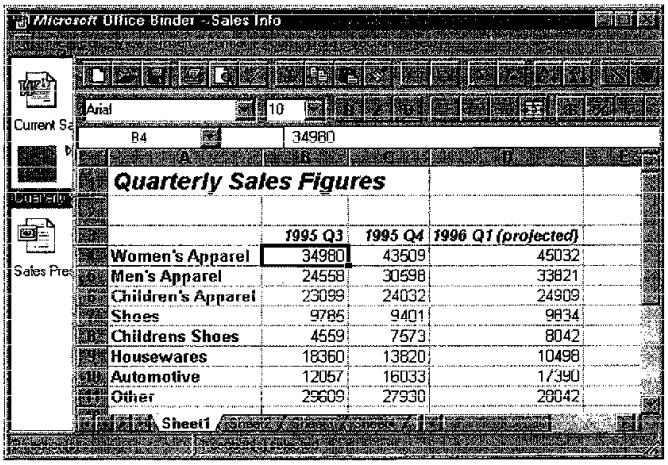
The Office Binder, a tool included with Microsoft Office 95, provides a good example of how this can be useful. The idea behind the Binder program is that a user might want to work in a unified way with information created by several Office applications. For instance, imagine a current sales report containing text created with Microsoft Word, quarterly financial data in a Microsoft Excel spreadsheet, and a sales presentation created with Microsoft PowerPoint. To collect these disparate kinds of information in a coherent whole, a user might embed the Excel spreadsheet and the PowerPoint presentation in the Word document. Another solution would be to embed all three in yet another document—in this case, in a binder.

The Office Binder lets a user work in a unified way with data from different applications

Figure 11-1 shows an example of the three kinds of data just described embedded in a binder document. As shown in the figure, the binder presents a two-part user interface. On the left appears an icon for each embedded document. On the right is the active document, the Excel spreadsheet. Each of the three documents in this binder is embedded, and the Excel spreadsheet is currently in-place active.

A binder document with three embedded ActiveX documents.

Figure 11-1



The ActiveX Documents technology builds on ordinary OLE documents

This binder document and the applications that have embedded data within it interact using conventions defined by the *ActiveX Documents* technology.[2] The binder is an ActiveX Documents container, while Excel, Word, and PowerPoint are all ActiveX Documents servers. Each application acts like an ordinary OLE embedded document server, although each one also has a little more functionality. For instance, any ActiveX Documents server is able to take over the entire editing area provided by the container. The container, which in this case is the binder document, essentially gets out of the way and lets the ActiveX Documents server completely control what the user sees. In Figure 11-1, for example, the user can have Excel take over the entire editing area by removing the window on the left containing the icons. An ActiveX Documents server presents a user interface that's more complete than the interface of an ordinary embedded document. To the user, in fact, it looks as if Excel is running independently—the limitations imposed on an in-place active embedded document are gone. Excel really is functioning as an embedded document here, as described in Chapter 8, but it can also offer extra features made possible by the ActiveX Documents technology. (And if you're starting to wonder what all this has to do with the Internet and the Web, be patient—it turns out to be very important.)

Describing ActiveX Documents

Supporting ActiveX Documents requires a few additional interfaces

All of the things required to present this richer user interface for an embedded document—the ability to take over the container's entire editing window, access to the server's print functions, and so on—are simply extensions to the current embedding and in-place activation features of OLE. Accordingly, containers and servers must first implement embedding and in-place activation

2 When Microsoft introduced the *ActiveX* designation, some technologies formerly assigned the *OLE* label were renamed. OLE Controls, for example, became ActiveX Controls. It's tempting to also assume that OLE Documents became ActiveX Documents, but this is not correct. The former OLE Documents technology is now referred to simply as *OLE*. ActiveX Documents describes a technology that builds upon this older technology—it's not just a new name.

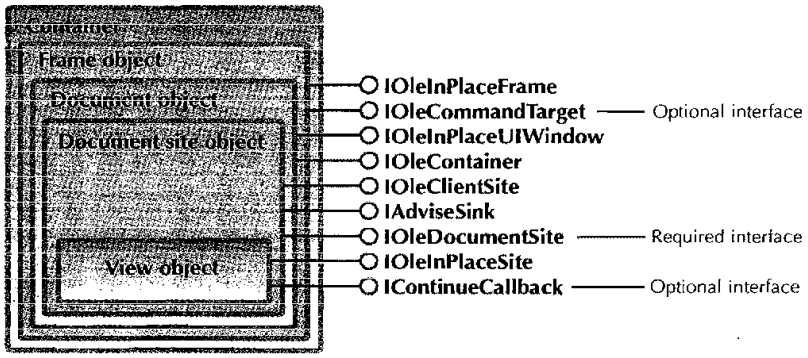
and then add a few more interfaces whose methods support the new features. The ActiveX Documents specification defines these extra interfaces.

ActiveX Documents containers must support IOleDocumentSite

Containers and servers To qualify as an ActiveX Documents container, an application must support all the interfaces OLE requires for embedding and in-place activation. As shown in Figure 11-2, ActiveX Documents containers must also support the IOleDocumentSite interface. This interface is implemented on a document site object, the ActiveX Documents analog of the client site object in an OLE container. An ActiveX Documents container provides one instance of a document site object for each embedded ActiveX document. A container can also support the IOleCommandTarget and IContinueCallback interfaces, both of which are discussed later in this section ("Commands," page 271).[3]

An ActiveX Documents container must implement at least one extra interface in addition to those required by OLE.

Figure 11-2



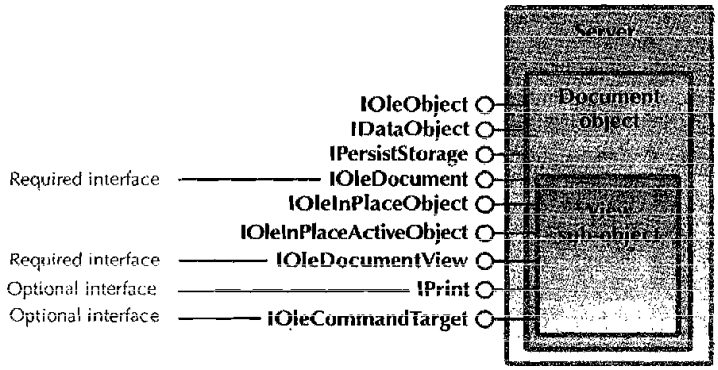
ActiveX Documents servers must support IOleDocument and IOleDocumentView

As shown in Figure 11-3 on the following page, acting as an ActiveX Documents server requires support for all the server-side embedding and in-place activation interfaces described in Chapter 8

3 Figure 11-2 includes one other interface, called IOleContainer, which allows a server to enumerate the objects managed by its container. Although this interface is not strictly required for an OLE container, IOleContainer turns out to be quite useful and so is commonly supported in the situations discussed in this chapter.

and more. A server might optionally support IPrint and IOleCommandTarget (discussed later), and it must support IOleDocument and IOleDocumentView. Understanding what these two mandatory interfaces do requires first understanding what the word *view* means in this context.

Figure 11-3 ***An ActiveX Documents server must implement at least two extra interfaces.***



A view acts like a filter for an application's data

An application such as Word or PowerPoint knows how to manipulate a certain kind of data. Word, for instance, works primarily with text, whereas PowerPoint works with slides and their contents. In each case, the application can present different views of its data. In Word, a user can see a document in Normal view, Page Layout view, or Outline view. PowerPoint allows the user to work with a presentation in Slide view, Outline view, Notes Pages view, and so on. Each view acts like a filter through which the user sees the application's data, each showing the same information in a different way.

Each view has its own sub-object

In an ActiveX Documents server, each view is represented by a view sub-object. This sub-object must implement the IOleDocumentView interface and might also implement IPrint and/or IOleCommandTarget. The server must also implement IOleDocument, which can be used to create view sub-objects. The container in turn implements one view object supporting IOleInPlaceSite (and perhaps IContinueCallback) for each view sub-object in the server.

Printing When a user prints a document directly from Word, what is actually printed depends on the view Word is currently displaying. If the user is looking at the document in Outline view, for example, the document's outline is printed. When a user prints a document from Word acting as an ActiveX Documents server, the same thing should occur. To allow this, a view sub-object can support IPrint. Using this interface, an ActiveX Documents container can ask a particular view sub-object in the server to print its view of the data. No longer does printing an embedded document mean that only the document's cached presentation is printed; with ActiveX Documents, the server itself can control exactly what is printed.

A view sub-object can implement IPrint to support printing

Printing can be a lengthy process, and users might get bored or change their minds about the wisdom of their print request. Once an ActiveX Documents container has asked a server to print a document, that server should periodically call the FContinuePrinting method in its container's IContinueCallback interface. This method's parameters include the number of the page currently being printed and the number of pages printed so far. A container might use these to keep its user apprised of the server's progress in printing. If the user tells the container to cancel the print job, the container can pass this information on to the server by setting an appropriate return code on FContinuePrinting. When the call returns to the server, it checks this code and, if necessary, cancels the print job.

A container can support IContinueCallback to keep informed about printing progress

Commands Both a container and a server can support the IOleCommandTarget interface. It's easiest to think of this interface as a stripped-down version of IDispatch. Recall that a dispinterface assigns DISPIDs to a group of methods and then lets a client invoke any method in that dispinterface using the single vtable method IDispatch::Invoke. The dispinterface itself is assigned a GUID, allowing the same DISPIDs to be used in different dispinterfaces without fear of ambiguity. With IOleCommandTarget, various *command groups* can be defined, each of which is assigned a GUID. Each command in a command group is assigned an integer value, analogous to the DISPIDs in a dispinterface. To execute

Both container and server can implement IOleCommandTarget to receive commands

any command, a client of IOleCommandTarget can invoke the Exec method of IOleCommandTarget, providing the GUID that identifies a command group along with an integer identifying a command in that group. It's also possible to pass a command with parameters using variants, the same mechanism used by IDispatch.

IOleCommandTarget is like a lightweight version of IDispatch

Why invent a new interface when IDispatch would certainly have sufficed? The answer is that the creators of ActiveX Documents felt that IDispatch was too heavyweight for the simple requirements here. The primary reason for using commands at all in this context is to allow a container to ask a server to perform such tasks as displaying its properties and to ensure that toolbar commands work as expected. Accordingly, an ActiveX Documents container and server typically exchange straightforward commands such as Open, Save, and Copy. Using IDispatch for such simple operations was seen as needlessly complex.

How the ActiveX Documents Technology Works

ActiveX Documents interactions are much like OLE interactions

Because the interfaces required for using ActiveX Documents are simply extensions of those already used for OLE embedding and in-place activation, the interactions between an ActiveX Documents container and server are very similar to those between an OLE container and server. As in OLE, an ActiveX Documents container (such as a binder document) loads an appropriate server (such as Excel or Word). The container then initializes the server using one of the IPersist* interfaces. A binder stores all its embedded documents' data in a single compound file, each in its own storage. This isn't the only choice, however. An ActiveX Documents container can also initialize a server from a file or from some other persistent storage, assuming that the server supports the appropriate IPersist* interface.

As part of the ordinary initialization process for an embedded document, a container invokes a server's IOleObject::SetClientSite method. In OLE, the container passes a pointer to its IOleClientSite interface as a parameter on this method. In ActiveX

Documents, however, the container passes a pointer to its IOleDocumentSite interface instead. From this, the server can determine whether its container views it as an ordinary OLE embedded object or as an ActiveX Documents object. When the user requests that an ActiveX document be activated by, say, double-clicking on it, an ActiveX Documents container invokes the server's IOleObject::DoVerb method as always. An ActiveX Documents server responds to this differently than an ordinary OLE embedding server does, however. When it receives a call to this method, the ActiveX Documents server invokes the only method in its container's IOleDocumentSite interface, the whimsically named ActivateMe, to request that its container make it active. The container can respond by using QueryInterface to ask the server for a pointer to its IOleDocument interface. The container then invokes IOleDocument::CreateView, which creates a new view sub-object in the server and returns a pointer to that object's IOleDocumentView interface. Using this interface's methods, the container can activate the view, work with it, and close it when it's no longer needed.

ActiveX Documents and the Web

ActiveX Documents objects were originally called Document Objects (DocObjects)

What does all this have to do with the Internet or the Web? Well, initially, nothing at all. ActiveX Documents objects were originally known as *Document Objects*, or just *DocObjects*, and they were first widely disseminated in the Office Binder program. The Binder is a useful tool, but it was created with a desktop-centric focus. What DocObjects provided, though, turned out to be useful in a much broader context. By supporting only a few extra interfaces in addition to those already required by OLE, one application could host another while still allowing a user to access the complete range of the hosted application's features. Those few extra interfaces brought with them the ability to work with everything the embedded application had to offer. The user could see a common frame yet work naturally within that frame with all types of data.

A web browser can provide an ActiveX Documents container

A binder document is one example of a common frame through which a user can access different applications. Another example, one that's much more interesting today, is a web browser. It too

can provide a common frame for accessing and working with all kinds of data and all kinds of applications. It was this realization—the tie to the Web—that prompted the name change from Document Objects to ActiveX Documents. As described next, the result was a complete revamping of Microsoft's web browser and ultimately of the Windows user interface itself.

Microsoft's Internet Explorer and COM

Although COM has since been applied to many other problems, it was originally created as part of a mechanism for creating compound documents. In some ways, the ultimate compound document is the World Wide Web. It shouldn't be surprising, then, that COM has been applied to the problem of web access, too.

Building a Browser from Components

Internet Explorer 3.0 relies on ActiveX Documents

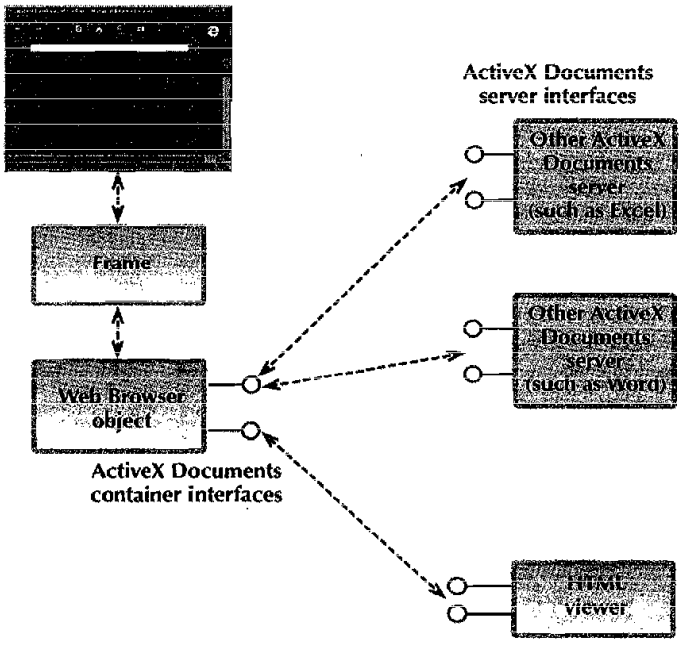
Think for a moment about what happens when a web browser downloads a typical HTML page from a web server. When the information is received, the browser interprets the HTML and displays the page to the user. In older browsers such as Microsoft's Internet Explorer 2.0, the code for displaying HTML pages was built into the browser itself. As browsers came to be used to display more than just HTML, however, they needed a general way to load code on demand to handle any kind of information. If the user downloads a file in Adobe Acrobat format, for instance, the browser must be able to load the correct code to interpret that file and display the information. ActiveX Documents defines this sort of relationship—one application acting as a frame for another. It makes sense, then, to build a web browser using this technology, which is exactly what's done in Internet Explorer (IE) 3.0.

IE 3.0's Web Browser object, an ActiveX Documents container, provides generic browsing functions

IE 3.0 separates generic browser functionality—navigating to a link, going forward and back, and so on—from the intelligence required to load, display, and manipulate particular kinds of information. The user sees one cohesive application, but the browser is actually built from several pieces, as shown in Figure 11-4. (Some of the relationships among the components are slightly

simplified in the figure.) The smallest piece is the Internet Explorer *frame*, implemented in IEXPLORE.EXE. This simple piece of code does little more than provide a host process for the Internet Explorer *Web Browser object* (once known as the *shell document viewer*), implemented in SHDOCVW.DLL. This object provides generic browser functionality, and it communicates with the frame through various COM interfaces (the details of which aren't included here). The Web Browser object has no knowledge at all of HTML documents or any other sort of displayable information. What it does know how to do, however, is to act as an ActiveX Documents container. By loading the appropriate ActiveX Documents server, the Web Browser object can let the user see and work with many different types of information.[4]

Microsoft's Internet Explorer 3.0 is built from separate components glued together using COM. ***Figure 11-4***

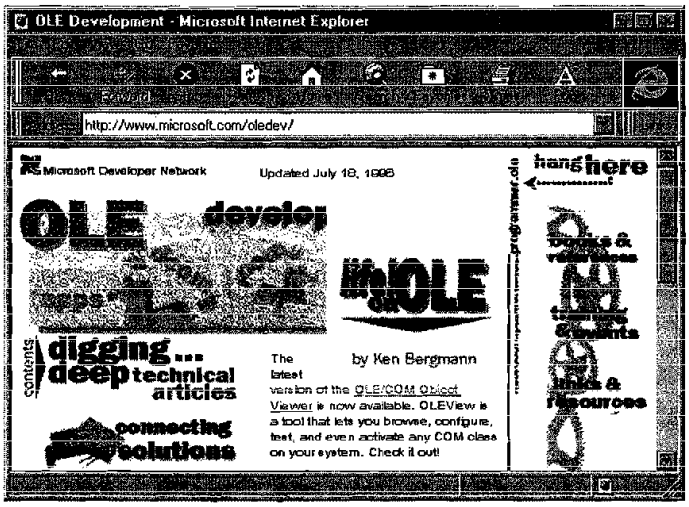


4 The Web Browser object also qualifies as an ActiveX control, which means that it can be plugged into any control container.

IE 3.0's HTML viewer, an ActiveX Documents server, knows how to display HTML

Its deconstructionist look notwithstanding, IE 3.0 is still a web browser, and a key part of its function is displaying HTML pages. When asked to display an HTML page, the Web Browser object loads the *HTML viewer*, shown in Figure 11-4. This viewer, implemented in MSHTML.DLL, is an ActiveX Documents server that contains all the code required to display and work with HTML documents. Figure 11-5 shows an HTML page displayed using IE 3.0's frame, Web Browser object, and HTML viewer. All these components work together to present the user with the familiar, seamless look of a web browser.

Figure 11-5 ***An ordinary HTML page displayed using Internet Explorer 3.0.***

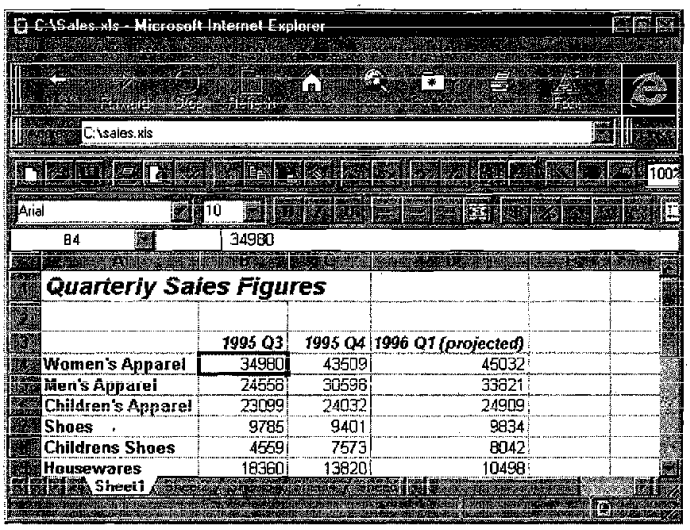


The Web Browser object can host any ActiveX Documents server

Because the Web Browser object is an ActiveX Documents container, it can also load and display anything that knows how to act as an ActiveX Documents server. Why not load an Excel file, for instance, into a web browser? Excel is capable of acting as an ActiveX Documents server, as shown earlier in the Binder example. Accordingly, IE 3.0's Web Browser object can load Excel and a spreadsheet the same way it loads the HTML viewer and an HTML document. Figure 11-6, an Excel spreadsheet displayed using IE 3.0, illustrates how this looks to a user. Because the ActiveX Documents technology exposes the full functionality of an embedded

application, hosting Excel within IE 3.0's Web Browser object in no way limits what the user can do. This spreadsheet can be directly edited just as if the user were working with a stand-alone instance of Excel.

An Excel spreadsheet displayed using Internet Explorer 3.0. ***Figure 11-6***



To the Web Browser object, both the HTML viewer and Excel look identical: they're ActiveX Documents servers. This ecumenical approach to browsing means that a web server can store information in various formats (not only HTML pages) and then let the browser load the appropriate code to work with that information. For example, if an Excel spreadsheet is stored on a web server on the Internet, an ActiveX Documents–enabled browser can let its user click on a reference to that page and then automatically load Excel (assuming that Excel or a simpler ActiveX Documents–enabled Excel viewer is available on the browser's machine) and display the spreadsheet as an ActiveX document within the browser. A user can now use one approach—browsing—to access HTML pages on the Web, application-specific files on a local hard drive, and nearly anything else.

The Web Browser object treats the HTML viewer, Excel, and any other ActiveX Documents server identically

Making the Windows Shell a Browser

The Windows shell provides a user interface

When you start Microsoft Windows, the user interface you see is provided by an application called the *shell*, which provides you with a way to access other applications and files on your machine. In Windows 95 and Windows NT 4, the standard shell presents a desktop metaphor, allowing you to work with the contents of your machine through folders and files in those folders. A web browser presents a different metaphor. Here you navigate through data and applications by following *hyperlinks* between documents, moving forward and back as needed. Given the popularity of browsing, integrating this new metaphor into the user interface is very desirable.

Given the structure of Internet Explorer, it's also very simple to accomplish. In IE 3.0, a generic browser (the Web Browser object) is loaded into a simple frame. Because that browser is an ActiveX Documents container, it allows users to access all sorts of information using the browsing metaphor. To let users access their systems as a whole using the browsing metaphor, then, all that's required is to modify the Windows shell so that it functions more like IE 3.0 and can then serve as a frame for the Web Browser object. In practice, this means adding support for a few more COM interfaces to the shell, not an especially onerous task. The shell itself can then host the Web Browser object in a natural way, and users can access information using this tool's generic navigation facilities. Files and applications on the local disk, a local network, or the Internet can all be browsed directly from the shell—there's no need for a special web browser application. And through the generic interfaces of ActiveX Documents, other applications can be loaded into that frame to work with other kinds of data, not just HTML pages.

Internet Explorer 4.0 extends Windows 95

This is exactly what happens in Internet Explorer 4.0. By supplying a new Windows shell, one that is capable of acting as a frame for the Web Browser object, the browsing metaphor can be applied throughout the user's environment. This is more than just a benefit for users—it's also a great example of the power of components. Code originally built for one application, a web browser, can be reused in a very general way.

Truly integrating browsing throughout the Windows user interface requires more than this, however. Browsing depends on the ability to create links among documents and to follow those links from one document to another. A traditional browser allows hyperlinks from one HTML document to another, but applying browsing more generally implies the ability to create more general links as well. A user might want to create a link from a PowerPoint presentation to a Word document, for example, or from a Word document to an Excel spreadsheet. Ordinary HTML hyperlinks aren't enough. To address this problem, the ActiveX family includes a technology called ActiveX Hyperlinks, which allows the creation of hyperlinks between all sorts of documents, not just HTML documents. The ActiveX Hyperlinks technology is already supported by IE 3.0's Web Browser object. (For details, see "ActiveX Hyperlinks," page 305.)

The ActiveX Hyperlinks technology allows the creation of hyperlinks among many kinds of documents

Making a Browser Programmable

Once the Windows shell itself lets you browse the Web, the need for a separate web browser application becomes less apparent. But while web browsers as such might one day fade into the mists of history, that day hasn't yet arrived. And even if browsers per se vanish, components such as the Web Browser object and the HTML viewer will survive. Like spreadsheets, word processors, and other applications, these components provide functions that are useful to other programs as well as to people. All that's required is for these components to expose a set of COM objects with appropriate interfaces that clients can use to access the components' services. In Internet Explorer 3.0, all of these interfaces are defined as dual interfaces, allowing easy access by clients written in Microsoft Visual Basic and similar languages as well as by C++ clients.

A web browser can expose its functions to applications as well as to people

Internet Explorer 3.0 has two components that provide programmability: the Web Browser object, providing generic browsing capabilities; and the HTML viewer, with its HTML-specific functionality. The Web Browser object is typically driven from the outside by, say, a Visual Basic program that uses this object to

The Web Browser object is typically driven from a tool such as Visual Basic

locate a particular document. To make this possible, the Web Browser object exposes methods that correspond to a user's actions, such as the following:

- The **Navigate** method is used to move to a new location specified by a hyperlink.
- The **GoBack** method is used to move to the previous location in the history list.
- The **GoForward** method is used to move to the next location in the history list.
- The **Refresh** method refreshes the current view by reloading the document.

The Web Browser object has methods and properties

Like most objects accessed through dual or dispatch interfaces, the Web Browser object also has properties. This object's properties include the following:

- The **Type** property returns the type of the currently loaded ActiveX Documents server, such as HTML or Excel.
- The **Busy** property indicates whether an activity such as a document load is in progress.
- The **Document** property returns a pointer to the IDispatch interface of the ActiveX Documents server for the currently loaded document. If an HTML document is loaded, for example, this property returns a pointer to the IDispatch interface of the HTML viewer. If an Excel spreadsheet is loaded, it returns a pointer to Excel's IDispatch interface. Using this pointer, a client of the Web Browser object can access the methods made available by the currently loaded ActiveX Documents server, whatever it happens to be.

The Web Browser object also has events

The Web Browser object can also send events, such as OnDownLoadComplete, an event indicating that the current page has been completely received. As with all events, the creator of a program driving the Web Browser object can write a subroutine that is called when this event is received.

Unlike the Web Browser object, the HTML viewer is typically driven from "inside." The viewer might, for example, load an HTML document containing an embedded script. This script then executes, making requests of objects within the HTML viewer as needed. The viewer supports several objects, arranged in a hierarchy.[5] A script can directly access the topmost object in this hierarchy, the Window object, and then acquire access to objects below it through the Window object's properties.

The HTML viewer is typically driven by a script in a loaded HTML file

The Window object represents the browser window that the user sees. Its methods include these three:

- The **Alert** method displays a simple message box.
- The **Prompt** method displays a message and prompts the user for a reply.
- The **Navigate** method causes a jump to a new location identified by a URL.

The Window object also has several properties, some of which return references to objects lower in the hierarchy. These properties include the following:

- The **History** property returns a reference to a History object containing a list of visited locations.
- The **Frames** property returns an array of the window's current frames.
- The **Document** property returns a reference to the current Document object.

Finally, the Window object is able to send two events: onLoad, sent when a page is loaded; and onUnload, sent (not surprisingly) when a page is unloaded.

5 The HTML viewer's object model is patterned after the model exposed by Netscape Navigator. This makes it straightforward to create scripts that work with both Navigator and Internet Explorer.

After the Window object, the Document object is probably the most important for creators of scripts. This object, located using the Window's Document property, represents the currently loaded HTML document. Its methods include Write, which writes text such as HTML code, and Open and Close, for opening and closing new documents. Among the Document object's many properties are bgColor, which sets a page's background color; linkColor, which sets the color for links on the page; and vlinkColor, which sets the color for links that the user has visited.

The Window object, the Document object, and all the other objects implemented by the HTML viewer can be accessed by scripts embedded in HTML documents that the viewer loads. If there were only one possible choice for a script language, it might make sense to build support for it into the HTML viewer itself. Several options for scripting languages are available, however, which suggests that a more general solution would be useful. That general solution, called ActiveX Scripting, is what the HTML viewer uses to execute scripts, and it's described next.

ActiveX Scripting

HTML documents can contain scripts written in languages such as JavaScript and VBScript

When the HTML viewer loads a document, that document might contain one or more embedded scripts. Those scripts can make use of the programmable objects exposed by the viewer, along with any objects that are loaded dynamically. Today the two leading languages for writing scripts embedded in HTML are Netscape's JavaScript and Microsoft's Visual Basic Script (formally known as Visual Basic Scripting Edition but commonly called VBScript). JavaScript is syntactically similar to the Java programming language, whereas VBScript is a subset of Visual Basic. It's not hard to imagine that other languages might be used for scripting as well.

A script is executed by a scripting engine under the control of a host

The HTML viewer itself has no reason to either know or care what language an executing script is written in. The script executes in a separate component called a *scripting engine,* while the viewer acts as a generic host for this engine. The viewer can instantiate a scripting engine, hand it a script, and tell it to begin executing the

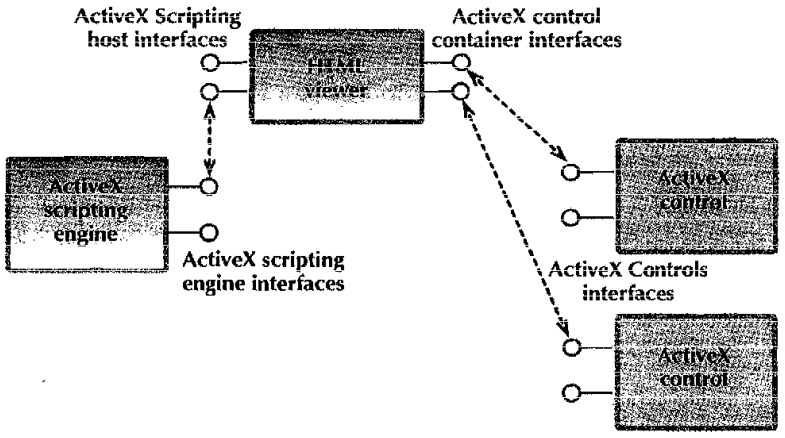
script. As the script executes, it can invoke methods in the viewer's objects and receive events from those objects. The interfaces supported by the HTML viewer and a scripting engine that make all this possible are defined by the *ActiveX Scripting* specification. (ActiveX Scripting was originally known as *OLE Scripting*.)

A host can provide built-in objects and might also load ActiveX controls

Furthermore, because the HTML viewer can also act as an ActiveX control container, loading an HTML page can result in loading one or more ActiveX controls as well as a script. Scripts executed by a scripting engine can interact not only with the built-in objects in the HTML viewer but also with any loaded controls. (In fact, an executing script can't distinguish between the two.) The relationships among the HTML viewer (acting as an ActiveX Scripting host and an ActiveX control container), a scripting engine, and a pair of ActiveX controls are shown in Figure 11-7. And finally, although this discussion uses only the HTML viewer as an example of an ActiveX Scripting host, this technology is in no way specific to this application. Any application can become an ActiveX Scripting host and then load and be driven by any scripting engine.

Figure 11-7

The HTML viewer is both a host for ActiveX scripting engines and a container for ActiveX controls.



Describing ActiveX Scripting

A script can access its host's objects

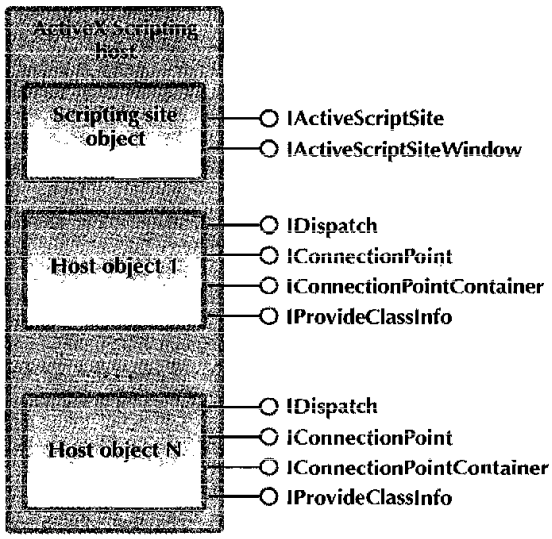
A scripting engine is a COM object, generally implemented as an in-process server, that is capable of executing a set of scripts—for instance, all those written in a particular language. Internet Explorer 3.0, for example, includes scripting engines for both VBScript and JavaScript. An ActiveX Scripting host typically implements objects whose methods, properties, and events can be invoked, accessed, and received by an executing script. For the HTML viewer, these objects include the Window object and the Document object, described in the previous section. The host can load objects such as ActiveX controls dynamically as well.

Scripting hosts must implement IActiveScriptSite and the host's objects must implement IDispatch

Figure 11-8 illustrates the objects and interfaces that can be implemented by an ActiveX Scripting host. As the figure shows, a host implements a *scripting site object* that supports the IActiveScriptSite interface. Using the methods in this interface, a scripting engine can acquire pointers to the interfaces of top-level objects the host makes available, inform the host of errors that occur, notify the host that the script has completed, and more. If the object supporting IActiveScriptSite provides its own user interface, it can also support IActiveScriptSiteWindow, allowing a scripting engine access to that object's window. Each object in the host, such as the Window and Document objects in the HTML viewer or a loaded ActiveX control, implements its own IDispatch interface, allowing a scripting engine to invoke its methods and access its properties. Each object should also implement IProvideClassInfo (or perhaps IProvideClassInfo2), allowing its client to access its type information. And finally, host objects that generate events also implement IConnectionPoint and IConnectionPointContainer.

Figure 11-9 illustrates the interfaces that a scripting engine can support. Every scripting engine must support the IActiveScript interface. A host uses the methods in IActiveScript to pass the scripting engine a pointer to the host's IActiveScriptSite interface, to tell the script to begin executing, and to perform other tasks. If the scripting engine can load scripts from persistent storage, it also supports one or more of the IPersist* interfaces, such as IPersistStorage, IPersistStreamInit, or IPersistPropertyBag. Scripting engines that

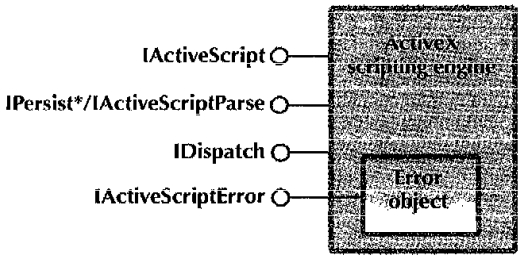
The interfaces that an ActiveX Scripting host and its objects can implement. **Figure 11-8**



allow script text to be added dynamically can support IActiveScriptParse, which lets a host such as the HTML viewer pass in a script received as part of an HTML file. If an error occurs during execution of a script, the engine passes its host a pointer to the IActiveScriptError interface, which is implemented by a distinct object in the engine. By calling methods in this interface, the host can learn more about the error. Finally, scripting engines that can accept events sent by a host or that allow a host to access the script's methods and properties must also implement IDispatch.

Scripting engines must implement IActiveScript and more

The interfaces that an ActiveX scripting engine can implement. **Figure 11-9**



An ActiveX Scripting Scenario

To understand how all this works, imagine that Internet Explorer 3.0's HTML viewer, an ActiveX Scripting host, loads the following very simple HTML document:

```
<HTML>
<TITLE>ActiveX Scripting Example</TITLE>
<BODY>
<H1>Illustrating Scripting</H1>
<SCRIPT LANGUAGE=VBScript>
  document.bgColor = "White"
  document.write "<HR>"
  document.write
    "Hello from the VBScript scripting engine"
  document.write "<HR>"
</SCRIPT>
</BODY>
</HTML>
```

The value of the HTML LANGUAGE parameter determines which scripting engine is loaded

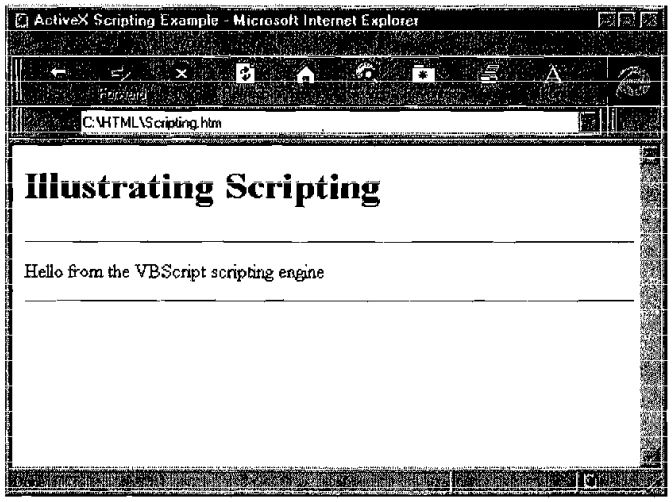
When the HTML viewer loads this document, it happily reads and interprets the first few lines using the HTML tags in the angle brackets. For example, the IE 3.0 viewer renders the line *<H1>Illustrating Scripting</H1>* as a level-one heading (based on the *H1* tag) as shown in Figure 11-10. When the viewer encounters the next line, however, beginning with the *SCRIPT* tag, it knows that it will need to load a scripting engine. Examining the *LANGUAGE* parameter, it determines that a VBScript engine is required. (If this were a JavaScript example, the value of the *LANGUAGE* parameter would be *JavaScript*.) The HTML viewer looks up *VBScript* in the registry—it's a ProgID, which is described in Chapter 4—and finds the associated CLSID. The viewer then calls CoCreateInstance with this CLSID to create an instance of the VBScript scripting engine and get an initial pointer to it.

The scripting host passes the text to the scripting engine

Once the engine is running, the host can acquire a pointer to the engine's IActiveScript interface. The host loads the HTML file's script into the scripting engine using methods in IActiveScriptParse and then invokes the scripting engine's IActiveScript::SetScriptSite method, providing a pointer to its own IActiveScriptSite interface. The basics are now in place for the host and the scripting engine to perform their complementary tasks.

The result of loading the example HTML file.

Figure 11-10



A host can pass the names of its objects to a scripting engine

The script that the engine is executing will need to use one or more of the objects supported by the host. Our VBScript example, for instance, sets the bgColor property and invokes the Write method of the HTML viewer's Document object. (Figure 11-10 shows the results.) To set properties and invoke methods of an object in its host, the scripting engine needs a pointer to that object's IDispatch interface. To allow the engine to get this pointer, the host can call IActiveScript::AddNamedItem for one or more of its objects, passing in the object's name as a character string. The HTML viewer, for instance, makes this call for the Window object, the top-level object in its hierarchy. This call isn't necessary for the lower-level objects in the hierarchy, however. Instead, the scripting engine can acquire pointers to these objects through properties on the Window object, as explained earlier.

A scripting engine can use an object name to ask a host for a pointer to that object

Once the host has informed the scripting engine about all the necessary objects, the host invokes IActiveScript::SetScriptState to tell the engine to begin executing its script. When the scripting engine needs a pointer to an object that it learned about through IActiveScript::AddNamedItem, it calls IActiveScriptSite::GetItemInfo with the name of that object. In our example, the scripting

engine calls this method only once, requesting information about the Window object. This call returns a pointer to the IUnknown interface of the named object. The scripting engine calls QueryInterface on the returned IUnknown pointer, asking for IDispatch. When it acquires the Window object's IDispatch pointer, the scripting engine next uses it to access this object's Document property and acquires a reference to the subordinate Document object. Using this reference, the scripting engine can set the Document object's bgColor property and invoke its Write method, as specified in the script.

Scripting engines receive events using the same mechanisms as control containers

A scripting engine calls its host to invoke methods and access properties. But a host might need to call a scripting engine, too, to inform it of events that have occurred. If an object in the host displays a button, for instance, the object might need to inform the scripting engine that the user has clicked on the button and that some event-handling code in the script should run. This is not a new problem—the process is just like an ActiveX control sending events to its container. Happily, the solution adopted by ActiveX Scripting is identical to that defined for ActiveX Controls. (In fact, the object sending the events to the script might actually be an ActiveX control.) Hosts such as the HTML viewer can provide type libraries for their objects, just as ActiveX controls do. A scripting engine gets a pointer to an object's type library and then reads the type library to learn how to build sinks for that object's events. This process is very similar to what control containers do (described in Chapter 9). And, as with controls, connection points are used to pass the necessary pointers from the engine to the objects to allow the events to be sent and received.

ActiveX Scripting allows a host to be transparently scripted from any language

By standardizing the interactions between an executing script and the objects it uses, ActiveX Scripting allows any host to work with any scripting engine. If the simple script shown earlier were written in JavaScript rather than VBScript, for example, nothing would change from the host's point of view, except that it would instantiate a different class of scripting engine. It's even possible to mix VBScript, JavaScript, and (potentially) other scripting languages in the same HTML file and have each script executed by its own

scripting engine. And, as mentioned earlier, ActiveX Scripting is useful for more than scripts loaded into a browser— scripting capabilities can be added to any application by implementing the interfaces required of an ActiveX Scripting host.

ActiveX Controls and the Internet

Internet Explorer 3.0 can load controls locally or from web servers

Visual Basic was the first widely used container for ActiveX controls, and its requirements were a driving factor in their original design. However, Internet Explorer 3.0's HTML viewer is a control container, too. An HTML page might contain data, for instance, that requires loading a specialized ActiveX control into the client's browser to view it. The code for this control might already be present on the client machine, or it might be downloaded from a web server when it's needed. Alternatively, IE 3.0 might download a platform-independent *applet* written in the Java language. (An applet is a program, typically a fairly small one, that runs inside a container of some kind, such as a web browser.) In either case, the result is the same: code is loaded as needed (perhaps from a web server) and executed on the browser's machine.

The arrival of the Internet led to changes in the ActiveX Controls technology

While the most visible effect of the Internet's collision with controls is probably their current name—ActiveX controls rather than OLE controls—the emergence of web browsers as control containers also caused some of the original, desktop-centric design decisions concerning controls to be revisited. As Chapter 9 explained, for example, the requirements a COM object must meet to qualify as an ActiveX control have been greatly reduced, a change made largely to accommodate the process of loading controls over slow Internet links. Loading potentially large amounts of data over those slow links has also led to extensions to the original controls technology. This section examines how ActiveX controls interact with browsers and discusses how a control can better fit into this new environment. Although Internet Explorer 3.0 serves as the example throughout, all the HTML shown here conforms to standards set by the World Wide Web Consortium —it contains nothing specific to Microsoft's browser.

Loading Controls into a Web Browser

An HTML document can cause ActiveX controls to be loaded using the *OBJECT* tag

Using the *OBJECT* tag in HTML, IE 3.0's HTML viewer can load and use any ActiveX control. And just as scripts can be written that use the objects built into the HTML viewer, so too can scripts make use of dynamically loaded controls. For example, suppose that the HTML viewer loads the following page:

```
<HTML>
<TITLE>HTML Control Example</TITLE>
<BODY>
<H1>Click An Arrow</H1>
<P>
<OBJECT
  CLASSID="clsid:B16553C0-06DB-101B-85B2-0000C009BE81"
  ID=SpinButton
  HEIGHT=200
  WIDTH=100
  HSPACE=85
>
</OBJECT>
<SCRIPT LANGUAGE=VBScript>
Sub SpinButton_SpinUp()
  MsgBox "(Up arrow clicked)"
End Sub
Sub SpinButton_SpinDown()
  MsgBox "(Down arrow clicked)"
End Sub
</SCRIPT>
</BODY>
</HTML>
```

A control does not need to do anything special to be loadable into IE 3.0

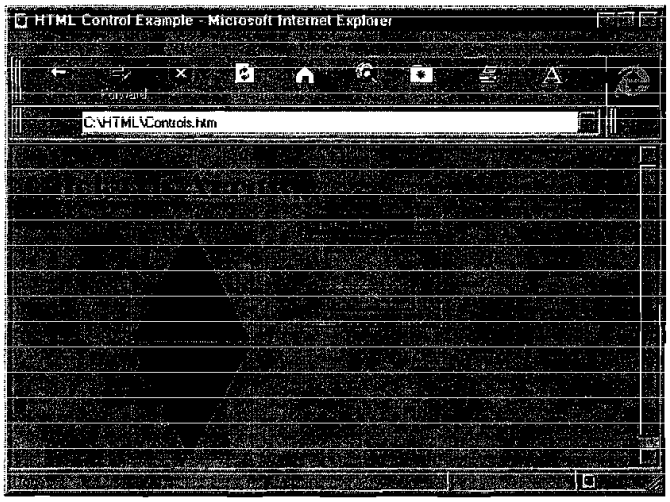
After the heading *Click An Arrow*, this document uses HTML's *OBJECT* tag to load an ActiveX control—in this case, it's the spin button control you saw in Chapter 9 ("An Application Developer's View," page 210). When this document is loaded, the HTML viewer reads the CLASSID attribute and then calls CoCreateInstance with that CLSID. The ID attribute gives the control a name that can be referred to in the script, while the remaining attributes determine the control's size and position on the page. There's nothing special about the control—this is the same code that was loaded into Visual Basic in Chapter 9's example. As in that example, the code is loaded locally, not from a web server. If no COM object is available locally with this CLSID, this example won't work.

Following the *OBJECT* tag is a simple VBScript program that again emulates the example in Chapter 9. Recall that the spin button control generates events when a user clicks on either of its arrows, events that can be caught when the control is loaded into Visual Basic. A VBScript program can also catch those events when the spin button control is loaded into the HTML viewer. (The previous section on ActiveX Scripting explained how the VBScript scripting engine is able to receive those events.) As the example shows, simple subroutines similar to those shown in Chapter 9 can be written in VBScript and executed when the user clicks on an arrow and the control generates an event. Figure 11-11 shows what a user sees after clicking on the up arrow.

An HTML page can contain scripts that use loaded controls

The result of loading the example HTML file and then clicking on the up arrow.

Figure 11-11



Loading a Control's Persistent Data

An ActiveX control usually has persistent data that it must load when it begins executing. Controls loaded from HTML files have several options for where this persistent data is kept and how it is loaded. This section describes these choices.

A control's persistent data can be stored directly in the HTML file

Loading small amounts of data How a control loads its persistent data depends on what kind of persistent data it has. Suppose, for instance, that a control has several properties whose values need to be set when the control is loaded. As shown here, those values can be stored in the HTML file itself using the *OBJECT* tag's *PARAM* element:

```
<OBJECT
  CLASSID="clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2"
  ID=label1
  WIDTH=150
  HEIGHT=500
>
<PARAM NAME="Angle" VALUE="270">
<PARAM NAME="Alignment" VALUE="2">
<PARAM NAME="Style" VALUE="0">
</OBJECT>
```

When IE 3.0's HTML viewer encounters this *OBJECT* tag, it loads the code for the specified control (which we'll again assume is already present locally) and requests a pointer to that control's IPersistPropertyBag interface. The viewer then reads the *PARAM* elements and hands their values to the control one at a time, as described in Chapter 5. (See "The IPersistPropertyBag Interface," page 125.)

A control's persistent data can be stored in a separate file

This approach works well with controls that can reasonably store their properties as text in an HTML file. But other controls might store their persistent data in a binary form and expect to load this data through IPersistStream. To allow this, the *OBJECT* tag can use the DATA attribute to specify a file that contains the data:

```
<OBJECT
  CLASSID="clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2"
  ID=chart1
  WIDTH=200
  HEIGHT=500
  DATA="http://www.acme.com/charts/profits.ods"
  >
</OBJECT>
```

When Internet Explorer's HTML viewer encounters the DATA attribute, it fetches the indicated file and hands it to the control as a stream through IPersistStream::Load. Although it's not shown here, it is also possible to place a limited amount of data for a control directly in the HTML file's DATA attribute.

Controls with large amounts of persistent data need to load it asynchronously

Loading large amounts of data Both examples shown so far work well with controls that have a relatively small amount of persistent data. But imagine a control whose persistent data includes large graphic or video files or other *binary large objects* (BLOBs). In this case, the control's BLOB data is certainly too big to be stored in the HTML file. It might also be impractical to store this data in the file named with the DATA attribute. Because the file would be loaded using IPersistStream, the file is handed to the control as a complete unit and all data in the file must be present on the local machine before the control can see any of it. Preventing the control from becoming even partially active until all the data has arrived is a less than optimal solution when that data is being loaded over a slow Internet link—users get frustrated when they're forced to spend much time looking at an hourglass icon.

A better approach would be to initialize the control with all its "small" persistent data and then load any BLOBs asynchronously. Web browsers do this today with ordinary HTML pages, first loading the page's text and then fetching any embedded images. The benefit is that the user sees an active (although incomplete) page almost immediately and gradually gets the larger data elements which complete the page. Controls with BLOB data can work the same way—first loading any smaller data and becoming at least partially active to the user before gradually loading BLOB data.

Controls can define data path properties

This two-part initialization scheme relies on *data path properties*. A data path property is like any other property a control might support except that its value can be a URL. Data path properties are stored in the file identified by the DATA attribute of the *OBJECT* tag and are passed to the control through IPersistStream. When the control receives its properties, it examines them individually

and uses their values to initialize itself. When the control recognizes a data path property, however, it can extract the property's URL and use it to locate and load the data it refers to.

A control's container typically participates in referencing data path properties

The URL contained in the data path property can be absolute, containing everything needed to locate the machine on which the data resides. For example, the data path property shown in Figure 11-12 contains an absolute URL. More likely, however, a data path property's value is a relative URL, which must be combined with a base URL (such as that of the page in which the control is embedded) to completely specify the data's location. Because only the control's container knows this base URL, the container is typically involved in the process of locating the data identified by a data path property. To allow this involvement, the container implements the IBindHost interface.

Figure 11-12 ***Three properties for a control, one of which is a data path property.***

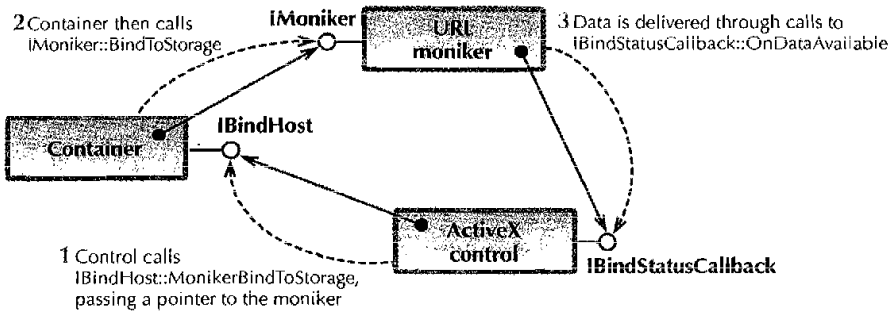
A container might need to give some controls' downloads higher priority than others

When a control needs to load information identified by a data path property, it can invoke its container's IBindHost::CreateMoniker method, passing in the URL contained in the data path property. The host creates a moniker (such as a URL moniker) that identifies the absolute location of the data and returns a pointer to that moniker back to the control. The control is then free to call the moniker's IMoniker::BindToStorage method to retrieve the information refer-red to by the data path property. Normally, however, a well-behaved control won't do this. Instead, it allows its container to participate in the binding process. The container, for instance, might have loaded several controls, each containing data path properties referencing remote BLOBs and all loading those BLOBs at the same time. The container might need to prioritize the order in which BLOBs are loaded, based on information only it knows.

Accordingly, rather than calling IMoniker::BindToStorage directly, a control typically calls its container's IBindHost::MonikerBindToStorage method, as shown in Figure 11-13, passing a pointer to the moniker received from the container. The container then calls this moniker's BindToStorage method. If the moniker in question is a URL moniker, as it usually is, the information referenced by the data path property (the control's BLOB) is now downloaded asynchronously into a stream provided by and accessible to the control. The URL moniker keeps the control informed of the arrival of new chunks of data by periodically invoking OnDataAvailable in the control's implementation of IBindStatusCallback. (The control passes a pointer to this interface as a parameter on MonikerBindToStorage, and the container passes it to the moniker through the bind context object, as described in Chapter 6; see "How Asynchronous Monikers Work," page 148.)

An asynchronous moniker informs a control when a new chunk of the control's data arrives

Moniker binding for a data path property. ***Figure 11-13***



The benefit of all this complexity is that a control with BLOB data can become at least partially active quickly and then load larger files in the background a bit at a time. This makes for happier users, who aren't required to wait for all the control's data to arrive before beginning to use that control. And should a control find itself loaded into a container that doesn't support IBindHost, it can attempt to fend for itself by converting its data path properties into monikers using MkParseDisplayNameEx and directly calling BindToStorage on those monikers.

Loading large amounts of persistent data asynchronously lets the control become partially active more quickly

A control can inform its container when all data has been loaded

A control with data path properties must take one more action, however. When downloading data using a URL moniker, the control eventually receives an indication from the moniker that all the data has been loaded. The control must then inform its container that initialization has been completed and that it is fully ready for use. To do this, the control can send the OnReadyStateChange event to its container. The control can also set the value of a property called ReadyState, which the container can use to query the control's state. Through this event and/or property, the control can indicate different states: it has loaded all properties except asynchronously loaded BLOBs, it has loaded all properties including BLOBs, and so on.

Controls supporting these features work better in the Internet environment

ActiveX controls such as the spin button control that were created before the advent of these new Internet-related technologies don't take advantage of these new features. Although older controls can be loaded and used by control container web browsers, they don't provide all the benefits of a control written with the Internet in mind. Controls that are Internet-aware are made more efficient with support for data path properties and asynchronous downloading along with the OnReadyStateChange event and/or the ReadyState property. These features are by no means required, but they make a control much better suited for use inside a web browser.

Downloading Controls

A control's code can be downloaded when needed from a web server

In the examples shown so far, a control's data might have been stored on a remote machine, but the code for the control was assumed to be resident on the browser's machine. This need not be the case, however. Why not load a control's code from a web server when it's needed? To tell the browser where the code is, the *OBJECT* tag can include a CODEBASE attribute. Here's a simple example:

```
<OBJECT
  CLASSID="clsid:B16553A0-06DB-101B-85B4-0000C009BE05"
  CODEBASE="http://www.acme.com/welcome/mapshow.ocx"
  DATA="http://www.acme.com/maps/campus.geo"
```

```
  ID=MapDisplay
  HEIGHT=450
  WIDTH=450
>
</OBJECT>
```

HTML's CODEBASE attribute indicates where the code resides

When Internet Explorer 3.0 encounters this tag in an HTML file, it downloads the file named by the CODEBASE attribute (assuming that no code for this CLSID is currently present on the machine) and then instantiates the control. In this example, the referenced object is an ActiveX control, but Internet Explorer 3.0 also supports the downloading of Java applets. An attribute called CODETYPE on the *OBJECT* tag can be used to indicate the MIME (Multipurpose Internet Mail Extensions) type of this object, such as *application/java-vm*, which lets the browser decide whether it's worthwhile to download it.[6] And as the example shows, it's also legal to use the CODEBASE and DATA attributes at the same time, causing the browser to download both a control's code and its persistent data.

Three main options are available for packaging downloaded code

How downloading works In some cases, all that's required to download code is to copy a single executable file from a web server to the browser's machine. In other cases, it might be necessary to copy more than one executable file along with one or more supporting files. To deal with this variability, the *Internet Component Download* service used by Internet Explorer defines three packaging schemes for downloaded code:

- A portable executable (PE), containing a single executable file with an extension such as OCX, DLL, or EXE.
- A cabinet file, identified by the file extension CAB. A cabinet file can contain one or more executables, all compressed into a single package and downloaded as a unit. It also includes an INF file that directs the installation process of the cabinet's files.

6 MIME types are used throughout the Web environment to indicate data types. Other commonly seen MIME types are *text/html*, *image/gif*, *image/jpeg*, and *video/mpeg*. At the time this book is being written, no permanent MIME type has yet been defined for ActiveX controls.

- A stand-alone INF file, containing only references to other files that should be downloaded. An INF file can contain URLs referring to files on a single machine or on several machines. It can also specify options for which files to download depending on the type of client platform making the request. For example, a request to download an INF file made from a Windows 95 system and the same request made from a Macintosh system might result in copying different binaries.

The filename specified in the CODEBASE attribute can optionally be followed by a version number. If it is, the file is downloaded only if this version number is more recent than any version of this file currently resident on the system.

A call to CoGetClassObjectFromURL does everything required to download and install a new component

When a browser such as Internet Explorer attempts to download the code for an ActiveX control, its real goal is to create one or more COM objects using that code. Ultimately, then, the browser must acquire a pointer to the IClassFactory interface of the control's class factory and call CreateInstance. The Internet Component Download service makes this very easy. When Internet Explorer encounters a CODEBASE attribute inside an *OBJECT* tag and decides to download the associated code, it needs to call only the single function CoGetClassObjectFromURL. Like CoGetClassObject (discussed in "Using a class factory," page 61), this function returns a pointer to a class factory. As its name suggests, the caller passes in a URL specifying where to find the code. This URL can name a portable executable, a cabinet file, or an INF file, and the browser takes this value directly from the CODEBASE attribute in the *OBJECT* tag. The caller can also pass in the CLSID from the tag's CLSID attribute or the MIME type of the object indicated by the CODETYPE attribute. (The MIME type is mapped to a CLSID using the system registry.) Making this single call causes the control's code to be copied to the browser's system (if it's not already present), verified as safe using WinVerifyTrust (discussed in the next section), and registered with the system registry. Once everything has been installed locally, CoGetClassObjectFromURL calls CoGetClassObject to return a pointer to the class factory of the new object.

Of course, the actual process is a bit more complex. The implementation of CoGetClassObjectFromURL relies on a URL moniker to accomplish the downloading, which means that a client making this call must implement IBindStatusCallback to receive progress notifications. The caller of CoGetClassObjectFromURL must also implement the ICodeInstall interface. This simple interface lets the client learn about any problems that crop up during the download and handle any necessary user interface issues. Also, once a component is downloaded, no automatic mechanism deletes it—it remains on that system's disk indefinitely. For the most part, however, clients such as Internet Explorer are shielded from the messy details of downloading objects.

Some mechanism must exist to guarantee the security of downloaded code

Ensuring the security of downloaded components Being able to download components as needed is a useful capability. By default, a downloaded Java applet is wrapped in a secure cocoon during execution. Because each applet has its own safe "sandbox" to play in, providing security in this way is sometimes called *sandboxing*. Unlike Java applets, however, ActiveX controls are binaries executing directly on the machine's hardware. Although ActiveX controls have capabilities that sandboxed Java applets do not, they also offer more opportunities for mischief. A malicious developer could easily create controls that, say, reformat the hard drive of any machine that installs them. If users can't have faith that a given control won't damage their system, they can't take the risk of downloading and running that control.

A trust provider can provide that guarantee

Creating that faith is the goal of the *Windows Trust Verification Services*. Through the single function call WinVerifyTrust, a user of this service can access one or more *trust providers*. In general, a trust provider can answer questions about whether a component can be trusted according to certain criteria. The initial release of this service includes only one choice, the Windows Software Publishing Trust Provider. This trust provider is able to answer the question that most concerns the potential user of a downloaded ActiveX control: was this control produced by someone I trust?

The goal is to ensure that the code is from a trusted supplier and has not been modified

At first glance, this might appear to be the wrong question. What users really want to know is whether this control will damage their system, not who created it. Unfortunately, there's no general way to determine this. The best users can do is assure themselves that the software was created by a trusted source and that it hasn't been modified since its creation. This is similar to the faith users express when buying packaged software. If the box carries the name of Lotus or Microsoft or another reputable vendor, and if the shrink wrap on the package isn't broken, users can feel confident that the software inside won't intentionally damage their system. Providing this same kind of confidence is the goal of the Windows Software Publishing Trust Provider.

A downloaded component can be digitally signed

When Internet Explorer 3.0 downloads a component, that component might carry with it a *digital signature*. A digital signature is a byte string that can be used to verify that the associated information was actually provided by a specific entity. More than that, a digital signature also verifies that the information (in this case, the downloaded code) hasn't been modified since the signature was affixed. In essence, the signature plays the role of both the company name on a software package and the package's shrink wrap.

A component carries a certificate to allow verification of its digital signature

To allow others to verify its digital signature, a component carries with it another byte string called a *certificate*. When Internet Explorer calls WinVerifyTrust, it passes in references to both the newly downloaded control's digital signature and its certificate. The trust provider examines both and returns an indication of success or failure.[7] If the check fails or if the component is from an untrusted source, IE 3.0 informs the user and offers a choice of whether to proceed. Note that because a digital signature verifies that the associated information hasn't been modified from its original form, it's impossible to silently insert viruses into the code. By having the developer add a signature to a component and having the

7 The details of how digital signatures work are beyond the scope of this book. For those who are familiar with the technology, the Windows Software Publishing Trust Provider uses PKCS #7 and X.509 version 3 certificates. For those who aren't, well, you can trust me on this.

browser check that signature after downloading, a system is created whereby a user can have a high degree of faith in the component's trustworthiness.

ActiveX and Java

While the length of time required to move from abstract concept to widespread deployment in software hasn't changed radically (writing code still takes time), the interval between development of a new concept and widespread assimilation of that concept certainly has. No technology better demonstrates this change than Java. Created by Sun Microsystems, Java is a programming language, one not too different from C++. But Java is more, too, offering exciting possibilities for the Internet and for COM.

Java programs are executed by the Java VM

As with most programming languages, it's possible to compile a program written in Java and produce a binary executable. This isn't commonly done today, however. Instead, Java source code is usually translated into a machine-independent *bytecode* rather than a machine-specific binary. This bytecode is then interpreted by the Java Virtual Machine (VM), software running on a real machine. Using this scheme, the same Java code can be executed on any machine that supports the Java VM.

A Java applet can be executed on any machine with Java VM software

One popular use of Java is to create applets, relatively small Java programs that run inside a container such as a web browser. Since Java applets can be distributed as bytecode rather than as machine-specific binaries, the same applet can be downloaded and executed on different systems. All that's necessary is for the target machine to have Java VM software available. It's also possible to create stand-alone applications in Java. Unlike applets, applications don't assume the existence of a container.

Java and COM

Java fits very well with COM

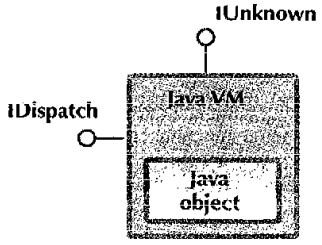
Microsoft has wholeheartedly endorsed the Java language. Microsoft's Java development tool, Visual J++, allows the creation of both applets and applications. At first glance, it might not be obvious

why Microsoft would choose to support this new language so strongly. After all, Java was created by Sun, a direct competitor. Furthermore, the machine-independent nature of Java's bytecode has led many to suggest that this new development tool could weaken the dominance of Windows/Intel systems. Despite this, however, Java offers a benefit that's very attractive: it meshes exceptionally well with COM. Although COM is officially language neutral, it's fair to say that COM and its supporting technologies were designed with C++ and Visual Basic in mind. Remarkably, even though it was created in a completely separate environment by a competing company, Java actually fits with COM as well as or even better than these two languages. A key part of this fit is that Java objects, like COM objects but unlike objects in C++, can support multiple interfaces. This, together with a few other features, makes Java an excellent language with which to implement and use COM objects. While this sort of technical serendipity is more the exception than the rule, Java and COM really are a natural pair.

Microsoft's Java VM makes Java objects look like COM objects

Microsoft's implementation of the Java Virtual Machine integrates Java objects and COM objects. Part of this integration is that from the point of view of a COM client, the Java VM makes a Java object appear to be just another COM object. With Java applets, for example, Microsoft's Java VM automatically constructs a dispinterface containing all the applet's public methods. With other Java objects, vtable interfaces are created. These methods are then accessible to clients of this object through a VM-provided implementation of IDispatch, as shown in Figure 11-14. To complete the illusion, the VM provides an implementation of IUnknown for each Java object, allowing clients to acquire pointers to other interfaces the object supports. The VM also implements a class factory, allowing a client to treat Java objects like any other COM objects. The Java programmer creates objects as usual—nothing special is required. All the services necessary to make those objects look like COM objects are supplied transparently by Microsoft's Java VM.

Microsoft's Java VM lets a Java applet look like a COM object. **Figure 11-14**



Microsoft's Java VM also provides the reverse translation: from the point of view of a Java object, an external COM object looks exactly like a Java object. Again, this integration is achieved without making any changes to the Java language itself. Instead, the Java VM transparently performs the necessary translations to map between the two kinds of objects.

Microsoft's Java VM also makes COM objects look like Java objects

Java is an excellent tool for creating COM clients, as the Java environment offers services that make life significantly easier for COM programmers. For example, a programmer working with COM objects in C++ must always be aware of reference counting. For a C++ client, this means calling Release whenever an interface pointer will no longer be used. Java programmers need not concern themselves with reference counting, however. Instead, the Java VM notices when an object is no longer referenced and automatically deletes it, a service known as *garbage collection*. When Microsoft's Java VM notices that the "garbage" object being collected is a COM object, it simply calls Release on the object. Unlike C++ COM clients, the creator of a COM client in Java never needs to worry about keeping track of which objects are no longer needed and then releasing them.

For acquiring references to new interfaces on an object, Microsoft's Java VM even hides calls to QueryInterface beneath the Java language's built-in operators. A Java programmer writes the same code to access a new interface regardless of whether that interface is on a Java object or a COM object. (In fact, the Java programmer can't tell them apart.) For a COM object, however, the Java VM intercedes, silently calling QueryInterface on the object and returning

The Java VM hides calls to QueryInterface

the new interface pointer. Unlike C++ developers, COM programmers working in Java never need to make explicit QueryInterface calls.

Type information is used to map between Java and COM

In order to provide all the translations required to map between Java and COM, Microsoft's implementation relies on the information stored in a COM object's type library. And to further integrate Java into the COM world, Microsoft offers Java class libraries exposing key COM functions such as CoCreateInstance, along with access to monikers, Structured Storage, and more. Although neither Java nor COM was designed with the other in mind, the two fit together very well.

Java Applets and Internet Explorer 3.0

Using Java to create COM objects and to write clients that access COM objects is an appealing idea. By hiding some of the rough edges, Java makes using COM that much easier. But a key purpose of Java, creating downloadable applets that run in web browsers, has no intrinsic connection to COM. How does Internet Explorer 3.0 support this?

Internet Explorer 3.0 treats Java applets like COM objects

Since Microsoft's implementation of the Java Virtual Machine makes a Java object look like a COM object, supporting Java applets is no different than supporting COM objects. The Java VM is implemented as an ActiveX control included with Internet Explorer 3.0. To execute a Java applet, the applet is simply loaded together with this control. To a control container such as Internet Explorer's HTML viewer, the applet looks like any other ActiveX control. And Microsoft's ActiveX control implementation of the Java VM can execute any standard Java applet, not only those created using Microsoft Visual J++.

Java applets can now be used wherever ActiveX controls are used

Implementing the Java VM as an ActiveX control has broader implications, too. Since applets look like ActiveX controls, and since controls can be driven by scripts, Java applets can also be scripted. Using the ActiveX Scripting interfaces, VBScript, JavaScript, or another scripting language can be used to access the

methods exposed by an applet. Java applets can also work with other applets and ActiveX controls in the same page. Finally, because the Java VM ActiveX control makes any Java applet look like a control, an applet can be loaded into any ActiveX control container and behave just as if it were a control. Although Java applets have historically relied on web browsers as containers, they can now be used with other control containers as well.

HTML pages can include Java applets using either the *OBJECT* or *APPLET* tag

As with ActiveX controls, the *OBJECT* tag can appear in an HTML page to indicate that a Java applet should be downloaded. Internet Explorer 3.0 also supports the *APPLET* tag, an older mechanism for embedding Java applets in HTML pages. When Internet Explorer 3.0 encounters an *APPLET* tag, it internally converts it to an *OBJECT* tag with the CLSID of the Java VM's ActiveX control. Internet Explorer then loads the Java VM ActiveX control and passes it the *APPLET* tag's parameters. The control then does everything required to download and run the applet.

Java applets can be digitally signed

Once downloaded, a Java applet can potentially call other COM objects or native code on the system. Ordinarily, an applet is sandboxed, as described earlier, and so isn't allowed to make these calls. As with ActiveX controls, however, Internet Explorer 3.0 allows a Java applet to be digitally signed and to have this signature checked when it's downloaded. Assuming that the signature identifies a trusted source, the applet is permitted to call other COM objects and local code just as a trusted ActiveX control would. For example, because any COM object looks like a Java object to an applet, it's possible for a digitally signed applet to access the automation services that many applications provide. A Java applet might access Excel's built-in services, for example, as a Visual Basic program might do.

ActiveX Hyperlinks

Browsing depends on hyperlinks

Part of the reason for the tremendous growth of the World Wide Web is surely the appeal of its fundamental metaphor: browsing. The central notion underlying browsing is the idea of hyperlinks.

To a user, a hyperlink appears on the screen as colored or underlined text, or as a graphic element embedded in the page, or perhaps in some other way. Clicking on a hyperlink changes what the user sees. In some cases, clicking on a hyperlink in an HTML document might simply result in displaying another part of that same document. In other situations, clicking on a hyperlink results in loading an entirely new document.

The ActiveX Hyperlinks technology allows hyperlinks to be created among various kinds of documents

Most users like the browsing paradigm—it's easy to learn and powerful to use—and Microsoft intends to integrate it throughout the Windows and Windows NT user interface. Key to this is finding a way to provide hyperlinks between all kinds of elements, not just HTML documents. Why can't we create a hyperlink between, say, a Word document and an Excel spreadsheet? Rather than embedding or linking the two documents using the conventions of OLE, why not tie them together with a hyperlink as if they were HTML documents? This is the goal of *ActiveX Hyperlinks*. By enabling the creation of hyperlinks that reference all kinds of elements, including but not limited to HTML documents, and by wrapping this generality in standard COM interfaces, ActiveX Hyperlinks applies the browsing metaphor to a broad range of documents and applications.

Describing ActiveX Hyperlinks

ActiveX hyperlink objects contain a friendly name, a moniker, and a location string

An ActiveX hyperlink is a COM object that supports the IHlink interface. It also supports IPersistStream, allowing its persistent state to be saved to and loaded from a stream, and IDataObject, allowing its contents to be copied using drag and drop or the clipboard. Every ActiveX hyperlink object contains (at least) three key pieces of information:

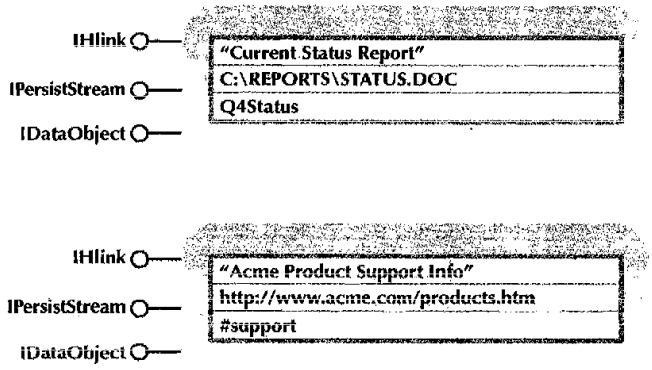
- A friendly name that can be displayed to the user when the hyperlink is visible. (Showing the friendly name is not required, however, because how a hyperlink is displayed is ultimately determined by the container that displays it, not by the hyperlink itself.)

- A moniker for the hyperlink's target—that is, for the application and data to which the hyperlink points.
- A string indicating a specific location within the target.

An ActiveX hyperlink might reference a location in a file or on a web page

For example, an ActiveX hyperlink to a Word file on a local machine might contain a friendly name such as *Current Status Report*, a file moniker that references the Word file, and a string indicating a location such as a Word bookmark within that file, as shown in Figure 11-15. An ActiveX hyperlink to an HTML document stored on the Internet might contain a friendly name such as *Acme Product Support Info*, a URL moniker that references the link's HTML document, and a string identifying a location within that document. A developer might use this second hyperlink to add an option to an application's help menu that directly connects the user to product support information on the World Wide Web.

Two example ActiveX hyperlink objects and their contents. ***Figure 11-15***



All ActiveX hyperlinks look the same to their clients, who see them primarily through the methods in IHlink. Those methods include the following:

- The **GetFriendlyName** method can be used by a client to learn the friendly name of the ActiveX hyperlink.
- The **GetMonikerReference** method returns the moniker and the location string from the ActiveX hyperlink.

- The **Navigate** method causes the ActiveX hyperlink to navigate to its target, the document to which it points.

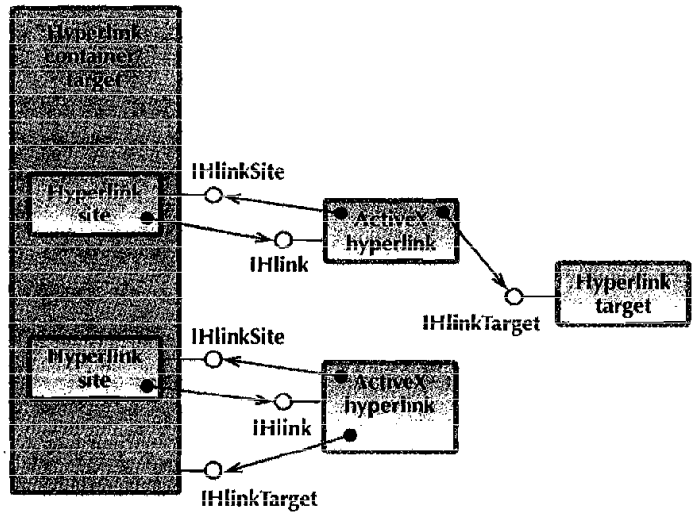
Standard library functions are used to create ActiveX hyperlinks

To create an ActiveX hyperlink object, a container need only call one of several standard library functions and pass in the appropriate data. HlinkCreateFromMoniker, for example, lets a container create a hyperlink object by providing the three required components of an ActiveX hyperlink: a moniker, a location string, and a friendly name. HlinkCreateFromString lets a container create an ActiveX hyperlink object by providing a location string, a friendly name, and a character string identifying the hyperlink's target.

ActiveX containers and targets implement IHlinkSite and IHlinkTarget, respectively

However it is created, an ActiveX hyperlink object communicates with its container through the container's implementation of IHlinkSite, as shown in Figure 11-16, and communicates with its target through the methods in IHlinkTarget. Note that a hyperlink can refer either to a location in the currently displayed document or to a location in another document. Supporting this first case requires the hyperlink's container to itself implement IHlinkTarget.

Figure 11-16 ***ActiveX hyperlinks, their targets, and a container.***



Although they aren't shown in the figure, two other components play a part in ActiveX hyperlinking: the *browse context object* and *hyperlink frames*. The browse context object supports the IHlinkBrowseContext interface, and it is responsible for maintaining the *navigation stack*. This data structure supports an integral part of the browsing metaphor: the ability to move forward and back in the list of visited documents. A traditional web browser maintains this list itself, but it applies only to hyperlinks between HTML documents. Because no single "browser" application might be able to encompass all the documents a user visits through ActiveX hyperlinking, an external object must maintain a list of visited documents. The navigation stack maintained by the browse context object generalizes the traditional web browser history list to include all documents browsed using ActiveX hyperlinks, including HTML documents, Word documents, Excel spreadsheets, or anything else.

The browse context object maintains a navigation stack

Finally, navigating to a hyperlink should ultimately result in displaying something new to the user. To provide some consistency, it's common (though not mandatory) to wrap a single frame around a succession of displayed documents accessed with ActiveX hyperlinks. Internet Explorer 3.0, for example, can be used to browse across many different kinds of data, and it gives the user a common frame for all of them. It can be useful to keep this frame informed about what's happening, allowing it to do whatever is needed to maintain a smooth look for the user. For example, all applications hosted within a hyperlink-aware frame can rely on the frame to locate the browse context object for them. To do this, a frame supports IHlinkFrame, whose methods are called by various components in the hyperlinking process at appropriate times. Internet Explorer 3.0's simple frame, IEXPLORE.EXE, implements this interface, as will Internet Explorer 4.0.

A hyperlink-aware frame provides a consistent environment for displaying a succession of documents

How ActiveX Hyperlink Objects Work

When a user clicks on a hyperlink, the container that receives the click creates an ActiveX hyperlink object containing the correct information and passes it a pointer to its IHlinkSite interface.

An ActiveX hyperlink object can refer to a location in the current document or in another document

(With the creation functions mentioned earlier, such as HlinkCreateFromString, all this can be done with a single function call.) Once the hyperlink object is running, the container calls its IHlink::Navigate method. To find out whether this hyperlink refers to another location in the document the container is currently displaying or to a location in another document, the implementation of IHlink::Navigate turns around and asks the container for a moniker to the container itself using IHlinkSite::GetMoniker. The ActiveX hyperlink object then compares this moniker with the moniker it already contains, the one naming the hyperlink's target. If the two monikers are the same, the hyperlink knows that it refers to another location in the current document. If not, it must refer to a location in a different document.

If a hyperlink object refers to a different document, it relies on its moniker to create that object

If the ActiveX hyperlink object determines that it refers to a location within the current document, the container for that document must support IHlinkTarget. (It's the target for this hyperlink, after all.) The hyperlink gets a pointer to this interface by calling the container's IHlinkSite::QueryService method. If this hyperlink does not refer to a location in the container's current document, the hyperlink object calls IMoniker::BindToObject on the moniker it contains. For a hyperlink containing a file moniker with a filename such as REPORT.DOC, for instance, calling BindToObject will typically start Microsoft Word (because of the DOC extension) and hand it this file through IPersistFile. If the hyperlink contains a URL moniker such as http://www.acme.com/report.htm, it will fetch the HTML page identified by this URL and hand it to a web browser such as Internet Explorer. Whatever kind of moniker is involved, the initial interface the hyperlink requests on BindToObject is IHlinkTarget.

The location string identifies a specific location within a document

One way or another, the ActiveX hyperlink object now has a pointer to the IHlinkTarget interface of the target. The hyperlink object next invokes IHlinkTarget::Navigate, passing in the location

string that this hyperlink stores. The source finds the correct information and causes it to be displayed.[8] If this hyperlink is to another location in the current document, the current window displays the new information. If necessary, however, a new window is created and correctly positioned to present a smooth transition to the user, much as is done with OLE in-place activation. And although this brief description omits the details, the frame (if there is one) is kept informed about what's going on, and the browse context object is updated with the result of this navigation throughout the process of following the hyperlink.

The Simple Hyperlinking API

The simple hyperlinking API makes all this easy to use

Integrating the browsing metaphor throughout their environment is likely to make users happy. Given what's just been described, however, it might leave software developers somewhat less pleased. Developers want a simple, powerful way to implement browsing, and although what we've seen so far is powerful, it's not especially simple. The implementor of a web browser or an application such as those in Microsoft Office might need a detailed understanding of the ActiveX hyperlinking architecture, but most programmers need only a straightforward way to add hyperlinks to their application. A simple hyperlinking API has been created to make this possible.

The primary purpose of this simple API is to make it easy to navigate to the target of a hyperlink. The API's small group of functions listed on the following page are focused around this goal.

8 This is similar to OLE linking using a composite moniker built from a file moniker and an item moniker. In that case, the file moniker identifies both the application and the document, while the item moniker passes the application a string that identifies a location within the document. By identifying a location within a document using a simple character string rather than an item moniker, the ActiveX Hyperlinks technology avoids the overhead of creating a moniker for the common case of hyperlinking to another location in the same document.

- The **HlinkSimpleNavigateToString** function causes a jump to another location, presenting the user with a new set of information. The caller passes in a string, such as a file-name or a URL, along with a location string and a few more parameters. The implementation of this call creates a moniker from the string (using MkParseDisplayNameEx, described in "A Generalized Approach to Naming," page 151) and creates an ActiveX hyperlink object containing that moniker and the location. It then navigates to the object this hyperlink identifies. A simpler version of this call, **HlinkNavigateString**, performs the same task but provides defaults for most of the parameters.
- The **HlinkSimpleNavigateToMoniker** function, like HlinkSimpleNavigateToString, causes a jump to another location. Its parameters are the same, too, except that the caller passes in a moniker instead of a string. A simpler version, called **HlinkNavigateMoniker**, provides defaults for most parameters.
- The **HlinkGoBack** function causes a jump to the previous location in the navigation stack maintained by the browse context object. This call works only if it is made by an application hosted in a hyperlink-aware frame, such as Internet Explorer. (This limitation exists because the implementation of this call relies on the frame to locate the browse context object—without it, there's no way to find the navigation stack and hence no way to go back.)
- The **HlinkGoForward** function causes a jump to the next location in the navigation stack. Like HlinkGoBack, it works only when made by an application hosted in a hyperlink-aware container.

Using these calls, any application can follow hyperlinks to any other application that supports the basic interfaces required to be a hyperlink target. An ActiveX control, for example, might present the user with a button that represents a link to a predefined Word

document. When the user clicks on this button, the control can call HlinkNavigateString with the name of the file, and a hyperlink jump to that document will immediately occur. Rather than understanding and implementing calls to the underlying objects and interfaces, a developer can achieve the most commonly used features of ActiveX hyperlinking with a minimum of effort.

Final Thoughts

New technologies force us to change

We work in a great business. Where else could new technology as transforming as that of the Internet and the World Wide Web so quickly become an important part of our lives? The downside of this enormous rate of change, of course, is that we're constantly forced to learn how to live with and use these new technologies. Sometimes this is easy. For the average software professional, learning to use a web browser takes less than five minutes. Sometimes, though, it's not so easy. Understanding the ActiveX technologies that underlie Microsoft's approach to the Web, for example, requires a firm grasp of COM, persistence, monikers, OLE, ActiveX controls, and more. It also requires understanding basic web technologies such as URLs and HTML. The reward for all this effort should be substantial, however. Whatever can be said about the tremendous amount of Internet hype—and it has frequently exceeded the bounds of rationality—one thing is sure: the Internet and the Web will be part of our lives for quite some time.

COM and the changes it has brought are here to stay

So, too, will ActiveX and OLE. COM and the technologies it has spawned have worked their way into the very fabric of Windows and Windows NT, two systems whose popularity is not declining. Understanding the ramifications of COM is essential to understanding software in the Microsoft world. And, one way or another, understanding the Microsoft world is important for nearly everyone in this exciting business we're in.