ActiveX Demystified

It's invasive. It's ubiquitous. But what exactly, is ActiveX?

David Chappel and David S. Linthicum

If you use a computer, Microsoft's ActiveX is probably part of your life. Whether you like it or not, the ActiveX technologies have become an essential part of Microsoft applications and tools; they are even finding their way into Microsoft operating systems. What impact is ActiveX having? How does it affect the millions of Microsoft-oriented developers and the tens of millions of Microsoft users? Answering these questions first requires addressing another question, one that turns out to be surprisingly hard to answer: Just what exactly is ActiveX?

First of all, get rid of the idea that the label "ActiveX" refers to some well-defined technology or group of technologies -- it doesn't. Instead, ActiveX is a brand name, like Calvin Klein or Ford. As with other brand names, what it's applied to can vary over time.

Still, the technologies grouped under the ActiveX umbrella aren't completely random. Many of them (but by no means all) are somehow related to the Internet and the Web. More important, all ActiveX technologies are built using Microsoft's Component Object Model (COM). But the obvious next step -- defining ActiveX to encompass all COM-based technologies -- is wrong. COM has found its way into almost everything Microsoft does these days, including Microsoft Office and Windows itself, and clearly these products aren't part of the ActiveX family. Annoying though it may be, we have to learn to liv e with this fuzzy, marketing-oriented notion of ActiveX -- it's the only one that's accurate.

But didn't ActiveX grow out of Object Linking and Embedding (OLE), Microsoft's compound-document technology? In fact, wasn't ActiveX just a new name for what was once called OLE? The answer to both questions is, "Well, sort of."

The story begins with OLE, a technology for creating compound documents. While OLE's first release was focused solely on compound documents, the next release, OLE2, introduced COM. COM grew out of the OLE architects' desire to provide a more general mechanism for allowing one piece of software to provide services to another. Accordingly, while OLE2 was the first technology to use it, COM isn't really tied to compound documents in any significant way. Very quickly, then, COM began to be used in technologies that had nothing whatsoever to do with compound documents.

So now Microsoft had a nice, general infrastructure technology -- but it's not a product -- and needed a brand n ame. The company's marketing wizards, perhaps unfortunately, chose to use "OLE" as that brand name. Deciding that the term should no longer be viewed as an acronym, Microsoft began adding the "OLE" tag to every technology that used COM. Most of these technologies, of course, had nothing at all to do with compound documents, so the company

spent several years trying to convince all of us that "OLE" no longer referred to just compound documents.

Then, in the spring of 1996, the company changed its collective mind one more time. A new brand name, ActiveX, was chosen, and "OLE" was again deemed to refer only to compound documents. And while OLE had once been a common brand name for nearly all COM-based technologies, COM had by now become so widely used that it was no longer possible to apply one name to everything. The result is today's undeniably confusing situation: ActiveX refers to a loosely defined set of COM-based technologies. OLE once again refers only to compound documents. An d COM, which was always the most important thing anyway, gets used more and more in the Microsoft world.

An important point to make here is that although the marketing label applied to many COM-based technologies has changed, COM itself has not. "The core spec for COM has been stable since 1993," according to Joe Maloney, COM group manager at Microsoft. "We've added additional functionality, but the definition of what COM is has remained consistent." Applications written against COM's initial release still work unchanged today.

The idea of an object model that's divorced from a programming language can seem odd. We understand what an object is in C++ or Java, but what's a COM object? A straightforward way to think about COM is as a packaging technology, a group of conventions and supporting libraries that allows interaction between different pieces of software in a consistent, object-oriented way. COM objects can be written in all sorts of languages, including C++, Java, Visual Basic, and more, an d they can be implemented in DLLs or in their own executables, running as distinct processes. A client using a COM object need not be aware of either what language the object is written in or whether it's running in a DLL or a separate process. To the client, it all looks the same.

Having such a general approach for packaging software turns out to be surprisingly useful. Two applications cooperating to give the user the illusion of a compound document, for example, can implement that cooperation as interactions between COM objects (which, of course, is exactly what happens with OLE compound documents today). Code that's downloaded from a Web server to run inside a browser can appear as a COM object to the browser, providing a standard way to package downloadable code (which is what ActiveX controls do).

Even the way an application interacts with its local operating system can be specified using COM (and new APIs for Windows and Windows NT are now often defined as COM objects). Despite its origin i n compound documents, COM can be usefully applied to a host of software problems.

Persistence

Creating a COM object by loading the right code is all very well, but is it enough? For some objects, the code is all that's really needed, but many objects also need the correct data loaded into them. These objects need to load their persistent data.

COM supports a number of persistence mechanisms. The simplest is file-based persistence, where an object just loads its persistent data from an ordinary file. For more complex situations, there's also a COM-based solution called structured storage . With structured storage, something analogous to a file system is built inside every file. Made up of storages , which are like directories, and streams , which are like files, structured storage allows many COM objects (possibly running inside many different applications) to share a single file.

When a client program creates a COM object, it's the client's responsibility to tell that object where to find its persistent data (if it has any). For COM objects that need to load persistent data, then, clients must do two things: Create the object itself, then tell it where to find the persistent data. COM supports other kinds of persistence, too, but ordinary files and structured storage are among the most commonly used.

Monikers

For many clients, creating and initializing a particular object instance is a perfectly acceptable thing to do. In some cases, though, it's just too much to expect a client to do this. An object might require very complex initialization, for example, or a client might need to use many different kinds of COM objects, each of which has its own idiosyncratic requirements for creation and/or initialization. To hide this kind of complexity, COM defines the notion of a moniker .

A moniker is a COM object like any other, but it has a special function: Each instance of a moniker object knows how to create and initialize exactly one other specif ic COM object instance. Monikers do what clients could do for themselves -- object creation and initialization -- but they hide the details from their clients.

But wait -- monikers are themselves COM objects, and they have their own persistent data (if they didn't, they'd have no idea what object they referred to). To use a moniker, then, a client must first create and initialize that moniker, then ask it to create and initialize the object it refers to. This seems patently stupid. Why can't the client just create the ultimate target object itself? What benefit does the moniker provide?

In many cases, the answer is "Nothing." Clients of COM objects often create and initialize those objects themselves, eschewing monikers entirely. But there are times when creating and initializing a COM object is so complex, so idiosyncratic, or just so painful that relying on a moniker can simplify a client's life. One example is connecting to a linked document in OLE, which was the first use of

monikers, but ther e are many others. Microsoft's Internet Explorer, for example, relies on monikers every time a user accesses a URL.

Automation

Like other kinds of objects, COM objects provide methods that their clients can call. Those methods are provided through interfaces that group methods into uniquely named collections. COM objects today can choose to expose their methods through two different kinds of interfaces. The first option, called vtable interfaces , works very well when the clients that will call those methods are written in C++. The second choice, called dispatch interfaces (usually shortened to dispinterfaces), works very well when clients are written in simpler languages such as Visual Basic (although they're also usable from C++ clients). For reasons related again to marketing, exposing methods using dispinterfaces has become known as automation .

The name "automation" was applied because of how dispinterfaces were first used. Developers of desktop applications wanted to allow other software to make use of their applications' functions. This is a situation tailor-made for COM, since its raison d'être is to allow one piece of software to expose its services to another. Because most of the code that would make use of those applications' services was expected to be written in Visual Basic, the applications' developers chose to expose their methods using dispinterfaces rather than vtable interfaces. And since doing this allowed writing programs that could automatically carry out, say, repetitive spreadsheet tasks that would otherwise have been done by hand, using dispinterface methods came to be known as automation.

Today, dispinterfaces are used in all kinds of situations, many of which have nothing to do with automating the use of a desktop application. Still, the name has stuck, adding one more confusing term to an area that already has more than its share.

Distributed Computing

COM's first incarnation assumed COM objects and their clients were running on the same machine (although they could still be in the same process or in different processes). From the beginning, however, COM's designers intended to add the capability for clients to create and access objects on other machines. Although COM first made its way into the world in 1993, Distributed COM (DCOM) didn't appear until the release of Windows NT 4.0 in mid-1996. Unquestionably an important part of the ActiveX family, DCOM is now available for Windows 95 as well (but don't hold your breath waiting for a Windows 3.1 version -- Microsoft says that's not going to happen).

DCOM really doesn't change much about how a client creates and interacts with a COM object. In fact, it might not change anything at all -- a client can use exactly the same code to access local and remote objects. In many cases, though, a client might choose to use a few DCOM extras (although these extras also work for local

objects -- COM's designers have worked hard to let clients remain unaware of where their o bjects are running). DCOM also includes a distributed security mechanism, providing authentication and data encryption. Windows NT 5.0, scheduled for release next year, will add support for Kerberos and other security protocols to DCOM. And to locate COM objects on other machines, DCOM today can make use of simple directory services such as the Domain Name System (DNS). Again, NT 5.0 will broaden the choices, adding support for Microsoft's Active Directory, which is based on DNS and Lightweight Directory Access Protocol (LDAP).

DCOM's traditional nemesis has been the Object Management Group's Common Object Request Broker Architecture (CORBA), which is embodied in many commercially available products, such as Iona's Orbix and Visigenic's VisiBroker. More recently, Java's Remote Method Invocation (RMI) has emerged as another choice for supporting distributed objects. Unlike CORBA and DCOM, both of which allow communication between objects written in various languages, RMI is focused on communication betwe en objects implemented in Java. This limitation certainly adds some constraints, but it also makes RMI very simple to use. Furthermore, RMI's developers had the luxury of designing their protocol specifically for Java, allowing them to make it an excellent match for the language's features. (COM, on the other hand, must deal with translations among the type systems of various languages, something that's almost never pleasant.)

Writing a DCOM server that can handle only a couple of clients is relatively straightforward. However, building a DCOM server that can effectively handle a couple hundred clients, or a couple thousand, is much more complex.

To make writing scalable DCOM servers easier, Microsoft has released the Microsoft Transaction Server (MTS). While MTS does provide support for transactions, it also provides services such as automatic threading and intelligent object reuse. Even applications that don't need transactions can benefit from using MTS since it makes writing scalable servers much easier; in fact, Microsoft encourages developers to write their MTS applications in Visual Basic, hardly a traditional choice for people creating industrial-strength servers. Every MTS application must be written as one or more COM objects, implemented in DLLs. To a client, MTS is typically invisible -- the client just creates and uses COM objects as always.

Standards for Components

Component-based application development holds the promise of building applications the same way we assemble electronics: out of prebuilt component parts. COM-based components for the desktop are known as ActiveX controls. (A very common terminological mistake is to confuse "ActiveX," a label for a broad family of technologies, with "ActiveX controls," a specific technology in that family.) An ActiveX control is just a COM object that follows certain standards in how it interacts with its client. For example, an ActiveX control must expose its methods via

automation, i.e., using dispinterfaces. This standardize d interaction allows the same control to be used in many different contexts. Behind its standard interfaces, an ActiveX control can do virtually anything, and controls implementing all kinds of functions are available from various software companies today.

ActiveX controls are written as DLLs, and so they must be loaded into some kind of container -- they can't run on their own. The archetypal container for ActiveX controls was Visual Basic (why do you think controls were required to use dispinterfaces?), but today there are many more choices. An especially important example of a control container today is Microsoft's Web browser, Internet Explorer. In fact, the realization that a Web browser could be a control container (and recognition that Java applets might otherwise come to own this market) caused Microsoft to significantly change both the technology and the name applied to COM-based desktop components.

What are now known as ActiveX controls were originally called OLE controls, and they were required to implement a large number of methods. This made them big, but so what? They were loaded off a machine's local hard drive into a container such as Visual Basic. Whether a control was a few hundred kilobytes or a couple megabytes made no significant difference, went the logic. But if a control were loaded into a Web browser, there was an excellent chance that that control would first be transferred across a slow phone connection to the Internet. Now, a control's size mattered crucially, and to require its creator to implement any more than the required minimum would needlessly increase its download time. Accordingly, at about the same time it changed the name, Microsoft decreed that what were now called ActiveX controls could implement only those features that were absolutely necessary for that control -- no more needless obesity was required.

For several years, the primary competitor to ActiveX controls was OpenDoc, promoted by Apple and IBM. Today, however, both of OpenDoc's sponsoring organ izations have officially declared it dead. Instead, most of the anti-Microsoft forces have lined up behind JavaBeans, a Java-based component architecture. Unlike controls, which are largely tied to Windows and distributed as machine-specific binaries, a JavaBean can run anywhere. The trade-off, of course, is that a Bean can't take full advantage of its local environment without compromising that portability. For many applications, such as writing a component that can be downloaded from the public Internet, JavaBeans is an excellent choice.

Today, there is a large and rapidly growing market for desktop components, nearly all of which are built as ActiveX controls (relatively few JavaBeans are available now). Standards for server components have been slower to arrive, however. On the desktop, Web browsers and programming environments such as VB and PowerBuilder are obvious choices for containers, but what should a server container be? Well, one excellent choice for a server-side component container is a t ransaction server -- and Microsoft is touting its own, MTS.

Microsoft's competitors are loath to see MTS and NT gain substantial ground. The most promising of their efforts to create a standard for server-side components is an extension of JavaBeans called Enterprise JavaBeans. This specification defines interfaces to a transaction server, not unlike MTS, and its supporters hope to convince independent software vendors to write their server components as Beans rather than COM components. Microsoft is ahead in this market -- MTS shipped in late 1996, while the Enterprise JavaBeans specification is quite new, and products supporting it aren't yet available.

The Future of ActiveX

It's fair to say that the ActiveX technologies will always be most at home in the world of Windows and Windows NT. But no matter how much Microsoft pushes its operating systems, most organizations will always have some diversity -- single-vendor environments just aren't in the cards. Accordingly, Microsoft is worki ng to make COM, DCOM, and some other parts of the ActiveX family available on other operating systems. Microsoft already provides ActiveX support for the Macintosh, including support for ActiveX controls (since Microsoft Office now depends heavily on COM, Microsoft had little choice but to support COM in that environment). Software AG is porting these technologies to various flavors of Unix and to IBM's OS/390 (the current name for the venerable MVS operating system). Digital and HP have also committed to providing these technologies on their systems, again by porting Microsoft's code.

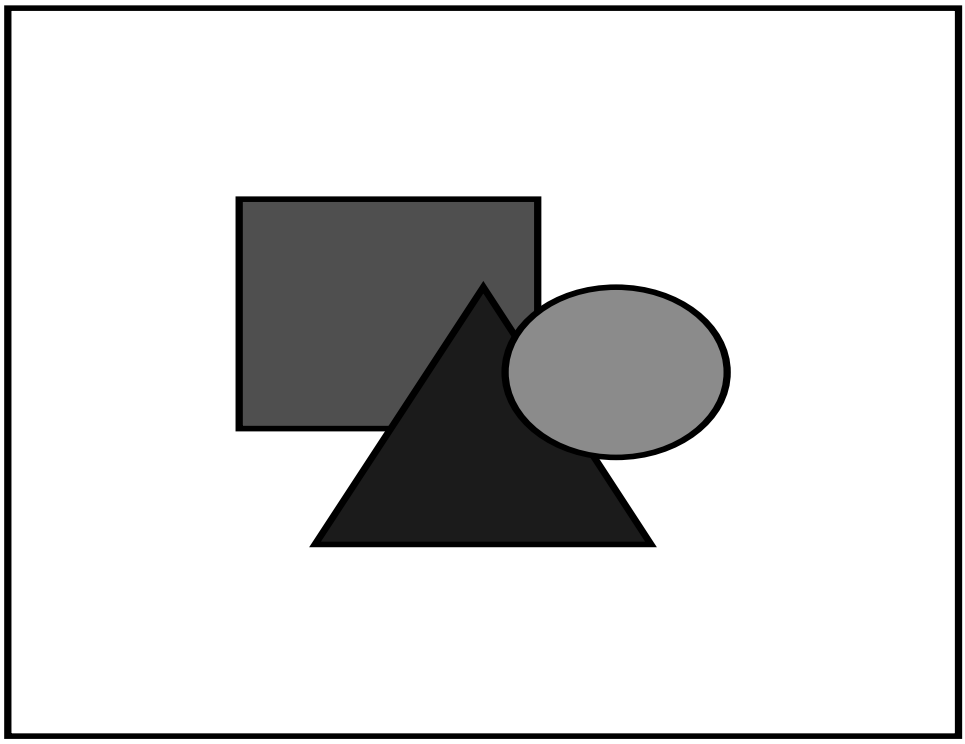
COM has become a crucial part of the software infrastructure in the Windows 95 and Windows NT environments. But there are still plenty of uncertainties about the future. Will Microsoft succeed, for example, in making COM a viable multiplatform technology? Fitting Windows NT servers into existing enterprises will all but require that DCOM and other distributed services be available on non-Microsoft platforms. The process h as taken longer than expected, and while many organizations have made promises in this area, not much code is actually shipping. Meanwhile, both CORBA-based products and Java's RMI are successfully running in multi-OS environments today. The more time passes before multiplatform DCOM becomes a reality, the larger CORBA and RMI's lead will become.

What about the contest between ActiveX controls and JavaBeans? Componentware is the next great wave in software development, whether that software runs in a Web browser or somewhere else. ActiveX controls are ahead today, but with the demise of OpenDoc, all of Microsoft's opponents have rallied behind a single competitor. If only because of users' desires to avoid monopoly, JavaBeans will very likely acquire some market share.

COM has grown to play a key role in Microsoft's Internet strategy, its applications, and even its operating systems. And as with all living software technologies, enhancements to COM are on the way. But whatever label is applied to the core COM-based technologies -- originally OLE, now ActiveX, and tomorrow

perhaps something else -- COM's importance shows no signs of declining. As long as Windows and Windows NT are important operating systems, the ActiveX technologies will play a significant role in our lives.

The Four Ways of ActiveX



The core technologies that enable ActiveX include COM/DCOM (the object model), ObjectRPC (the transport), the registry (a database of component locations), monikers, automation, and structured storage. All ActiveX components can be started the same way: The client calls CoCreateInstance. Unless the call specifies a particular remote machine, the call checks with the local registry database to locate the called component.

David Chappell (david@chappellassoc.com) is principal of Chappell & Associates, an education and consulting firm in Minneapolis. David S. Linthicum (linthicum@worldnet.att.net) is a senior manager with Ernst and Young's Center for Technology Enablement, in Vienna, Virginia.

Last Page