What OLE Is Really About

Kraig Brockschmidt
OLE Team, Microsoft Corporation

July 1996

Kraig Brockschmidt is a member of the OLE design team at Microsoft, involved in many aspect: continuing development and usage of this technology. Prior to holding this position, he was a s engineer in Microsoft's Developer Relations Group for three years, during which time he focuse OLE, versions 1 and 2, and produced the books Inside OLE 2 *and* Inside OLE, 2nd Edition *for M He has worked at Microsoft since 1988.*

Abstract

Microsoft has made and is continuing to make heavy investments in OLE-related technologies. been in development for more than seven years, and almost every new technology coming out somehow incorporates elements of OLE. Why does OLE deserve such an investment? And why significant for the independent software vendor (ISV) and the computer industry as a whole?

The answer, as this paper explores, is that Microsoft created OLE to solve, in an object-orientei component software, many practical problems encountered during Microsoft's lengthy experien systems and applications. OLE provides the necessary specifications and the key services that component software, which is ultimately a significant gain for the entire computing industry.

A Brief History of OLE

Graphical user interfaces popularized the "clipboard" metaphor--with "copy," "cut," and "paste' that greatly simplified the creation of a "compound document" that included text, graphics, an content. Prior to this invention, you had to print the text and graphics separately, then cut anc together with scissors and glue.

Although the clipboard works well for the initial creation of a compound document, Microsoft be its limitations in the late 1980s. Changing the text might require repositioning the graphics. Ec graphics required many difficult manual steps to get that data back into the original format, if possible at all. Microsoft's Applications division then created a complex dynamic data exchange to simplify these steps.

Out of the DDE protocol grew OLE version 1.0 (1991), which was made available to all develop standard. OLE 1.0 greatly enhanced the creation and management of compound documents: C "embedded objects" or "linked objects" in a document that retained the native data used to cre "link" to that data) as well as information about the format. (The acronym "OLE" is an abbrevia Linking and Embedding.") All the complexity of editing content was reduced to a double-click o presto! the object data was automatically brought back into the original editor.

But the OLE 1.0 designers realized that these "compound document objects" were actually a sp *software components*--small elements of software that can be "plugged in" to an application, tl extending the functionality of that application without requiring changes. In the compound doc

insert charts, sound, video, pictures, and many other kinds of components into that container
to update the container.

In the more general sense, component software has much broader applicability than to just co
documents. It is a multi-purpose "plug-in" model that is more powerful and flexible than other
dynamic-link libraries (DLLs), Visual Basic® controls (VBXs), and the like. This was the guiding
behind the design of OLE version 2.0, released in 1993. Not only did OLE 2.0 improve on the c
document facilities of OLE 1.0, but it built a vast infrastructure to support component software
of complexity.

The core of this infrastructure is a simple, elegant, yet very powerful and extensible architectu
Component Object Model, or COM. It is within COM that we find the solutions to some of the m
software problems, including those of extensible service architectures, "objects" outside applic
boundaries, and versioning. Furthermore, these solutions are concerned with *binary* componen
system rather than source-code components in an application. This article explores these probl
solutions that COM, and therefore the entirety of OLE, provides in the realm of running binary

What makes COM and OLE unique is that the architecture is one of *reusable designs*. There is
software industry about reusable *code*. Recently, with the new emphasis on *design patterns*, th
increasing interest in being able to reuse a design that still allows flexibility with implementati
we will see, one of the fundamental concepts in COM, that of an "interface," reflects the idea o
patterns.

COM and OLE therefore introduce a programming model based on reusable designs, as well as
implementation that provides fundamental services to make both design and code reuse possib
Microsoft has been introducing more and more "OLE Technologies" that build on the original ar
OLE 2.0, including enhancements to COM (Network OLE) and OLE-based technologies built into
system (shell extensions). In addition, this architecture has enabled groups outside Microsoft,
Microsoft's involvement, to create important technologies in the real-time market data, point-o
care, and insurance industries.

Of course, people have started to ask about OLE 3.0 and when it would be released, and wheth
shift from OLE 2.0 would be as major as the shift from OLE 1.0 (where the programming mode
completely changed). In fact, there is no OLE 3.0, nor are there plans for such a release. This i
reusable design idea: OLE's architecture *accommodates* new technologies--regardless of Micros
involvement--without requiring modification to the base designs. Thus, the name "OLE" has ce
acronym for "Object Linking and Embedding" and is simply OLE, pronounced "oh-lay" (as oppos
ee").

OLE, then, has moved from being a specific technology for a specific purpose to being a reusab
for component software that accommodates new designs and new technology.

So Just What Is OLE?

In the movie *The Gods Must Be Crazy*, an African bushman comes upon an empty Coke bottle
Westerner tossed from an airplane. The bushman believes the bottle is a gift from the gods, es
finds more and more uses for it: carrying water, digging holes, pounding stakes, making music
there any limits?

OLE is very much like the Coke bottle (except that at the end of the movie the bushman decid
bottle has caused too much trouble, so he tosses it off the edge of a cliff. This paper is not sug
same for OLE!). Just as the bushman would have a hard time pinning down exactly what the C
(and especially Microsoft's marketing groups) have a hard time pinning down a solid definition
Throughout its history, OLE has been promoted as many things, from Visual Editing to OLE Aut
Controls to Network OLE, and so on.

OLE might best be understood as a growth curve, like that shown in Figure 1. In other words, a

forward, OLE (based on COM) expands to accommodate new technology, never becoming obso
architecture. OLE can be described as an extensible systems object technology whose architect
accommodates new and existing designs.

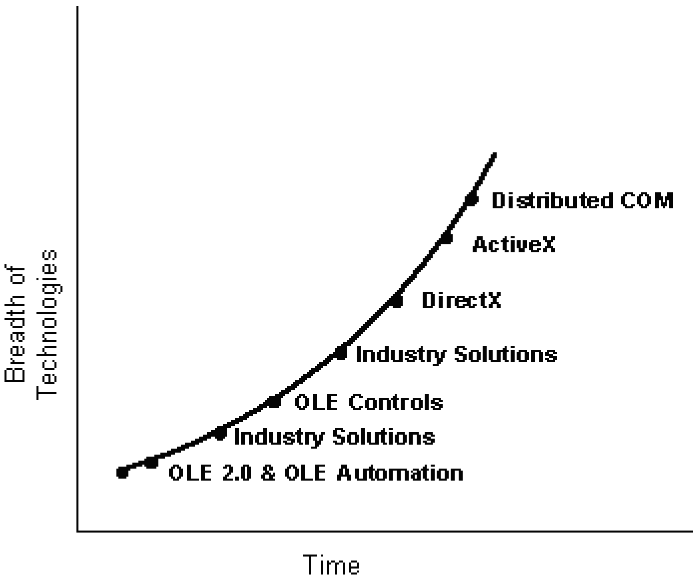


Figure 1. The OLE growth curve

A "systems object technology" means that one can work with object-oriented principles in the
running operating system--encapsulated, polymorphic, and reusable components exist and inte
binary entities, as opposed to source code definitions. New components, developed by anyone
be added into the running system, thereby immediately extending the services offered to appli
those applications are already running. This is what is meant by an *extensible service architec*
a number of different problems that are not relevant to source-code programming.

A newly installed operating system offers a basic set of services which developers employ in th
applications. COM and OLE provide the ability for anyone to extend the system with new servic
requiring a change to the operating system. That is, COM and OLE make it possible for anyone
services with which developers create more and more innovative applications. Furthermore, th
accomplished without requiring any kind of central control or coordination between vendors. M
than stifling creative competition has built the infrastructure that allows everyone to create co
isolation but still be able to integrate those components in rich ways.

It is this potential for integration that can lead to significant improvements in how we develop
most importantly, the end-user's experience with the computer as a problem-solving tool. The
this paper examines that potential in more detail. First, however, let's see how Microsoft's expe
operating systems and applications led to the designs of COM and OLE.

Problems and Solutions: The COM Architecture

How did Microsoft end up at OLE as an architecture on which it is betting its future? One must
the design of COM and OLE were not just dreamed up by a Microsoft architect who drank too m
night and stayed awake playing Reversi on "expert" level. The fundamental designs in COM are
many years of Microsoft's experience in the business of operating systems and applications.

In particular, Microsoft has repeatedly faced the problem of offering new software services for
applications. The primary concern of an operating systems vendor is *how best to provide these*
applications, because applications are built on system services and the system will not succeed
applications. Microsoft has had to deal with such issues throughout the lifetimes of MS-DOS® a

because such services are almost always shipped with the operating system. However, in a cor software environment, nearly all new components (that implement a service) are shipped sepa separate from the system but separate from each other. Component software requires an arch which any developer or vendor can deliver a component at any time and have that component immediately useful to applications on any given system.

To be used successfully, component software requires that applications always check on what exist when they need them, instead of assuming there is only a limited set. When a new comp to the system, it should become instantly available to all applications, even those that are alre example, consider a word processor that has a "Check Spelling" menu command whose implem on the existence of a suitable spell-checker component. If a newer and better component is ad system, that application can immediately take advantage of it the next time the user clicks tha

A system that supports component software must therefore support a generic "service abstrac architecture that defines how all types of components appear and how they are manipulated. I architecture must be extensible, so that a new component *category* (as opposed to an *impleme* existing type) can be introduced without having to revise the architecture. This is the problem *extensible service architecture*. For instance, it might be easy to define an architecture that ac components that provide content for compound documents, but can that same architecture acc specifications for custom controls? In other words, the architecture must expect that new com categories, will be defined later on.

The other big problem that such an architecture must solve is that of "versioning." It turns out definition of a component type is easy to manage, as is the first implementation of any particu The difficulty comes in managing revisions to the designs and the implementations over time. are the results of Microsoft's experience with such problems.

From DLLs to COM

Let's say you have some kind of software service you would like to provide, maybe a library of functions. The usual way this is accomplished is to implement a dynamic-link library (DLL) tha functions in the DLL. We call this code module the "server." The server then exposes the "flat because all the functionality of the service you are providing is described in a flat list of functic "client" application is one that calls these functions to perform various operations.

This flat API setup has been used for quite some time and is considered the *de facto* method o software service--and it works reasonably well for code modules that are shipped with an opera Microsoft has also used the technique for creating VBXes as well as Microsoft® Foundation Clas DLLs. But as we will see in this section, there are many hidden problems that arise when new installed separately on a machine over a long period of time. We will see that the basic DLL mc component software, and that COM was designed to overcome the problems noted above.

As a starting point, let's use the spell-checker component mentioned in the last section. Assum has written a specification that describes what a "spell checker" component does--that is, what a component that falls into this category. This specification describes the *service* of type "spell might be as simple as follows:

> A spell-checker component is a DLL that exports one function to check whether or not a word exists in the current dictionary. The function prototype is as follows:
>
> `BOOL WINAPI LookUpWord(wchar_t *pszWord)`
>
> This function must be exported as ordinal number 10.

Now, anyone can sit down and create an implementation of this service category, producing a

DLL is loaded into a process we just call it an "object." This is saying that the relationship betw component/server instance and the service category is just like that between an object instanc definition in a language like C++.

Let's say that hypothetical vendor Basic Software creates a server called BASICSPL.DLL and se hypothetical ACME Software, Inc. ACME incorporates this DLL into their text application, Acmel implementing its Tools/Spelling... menu command by parsing each word in the text and passin one into **LookUpWord**. The code for this command might appear as follows:

```
void OnToolsSpelling(void)
    {
    wchar_t *pszWord;
    pszWord=GetFirstWord();  //Retrieve first word in text.
    while (NULL!=pszWord)
        {
        if (FALSE==LookUpWord(pszWord))  //Check if word is in dictionary.
            [Alert user]
        pszWord=GetNextWord(pszWord);  //Go to next word in text.
        }
    return;
    }
```

When an application uses a function provided by some DLL, that application must have an abs name to identify exactly which function it wants to call. In the basic DLL model on Windows, al function identities include the code module plus the exported function ordinal (or text name). the **LookUpWord** function (ordinal 10) in BASICSPL.DLL is known as **BASICSPL.DLL:10**.

When the ACME developers compile and link AcmeNote, the call to **LookUpWord** is stored in t "import record" for **BASICSPL.DLL:10**. (An "import library" is what provides the mapping fron **LookUpWord** as known to the compiler and the import record entry of SPELLCHK.DLL:10. Do **dumpbin /imports** on some .EXE and you'll see a list of module:ordinal records for all the dyn import record is a "dynamic link" to the function in the DLL because the actual address of the f known at compile time. Each **module:ordinal** record is essentially an alias for an address that until run time. That is, when the kernel loads the application, it loads the necessary DLLs into prior to calling **WinMain**), thereby giving each exported function a unique address. The kernel each import record with a call to that known address.

An application can perform these same steps manually to control when the DLL gets loaded. O this is to bypass the kernel's default error handling when the DLL or exported function does no run AcmeNote and BASICSPL.DLL can't be found, the system will complain with some wonderf "The dynamic link library BASICSPL.DLL cannot be found in the path <list the PATH environme followed by "The application failed to initialize properly (0xc0000135). Click OK to terminate th Uh-huh. Yeah. By having the user perform these steps manually, AcmeNote could inadvertentl Tools/Spelling... command if the DLL isn't found.

If the kernel can find the DLL but cannot find the exported function, it says other nasty things. 3.1 kernel puts up an ugly white system-modal dialog that says, "Call to undefined dynlink." C Windows NT kernel says, "The application failed to initialize properly (0xc0000139). Click OK t application." To an end user, *things are definitely NOT OK* and they will probably be calling cus very soon.

In the case of AcmeNote, we can make the application boot faster if we don't load the DLL at a actually execute the Tools/Spelling command. To do this, we explicitly load the DLL with the **L** Win32® API, then retrieve the address of **LookUpWord** (ordinal 10) using the **GetProcAddre** We then call the function through this pointer:

```
//Function pointer type for LookUpWord
typedef BOOL (WINAPI *PFNLOOKUP)(wchar_t *);
void OnToolsSpelling(void)
    {
```

```
    HINSTANCE  hMod;
    PFNLOOKUP  pfnLookup;
    wchar_t   *pszWord;
    hMod=LoadLibrary("BASICSPL.DLL");
    if (NULL==hMod)
        [Spell-checker not found, show error]
    //Get address of ordinal 10
    pfnLookup=(PFNLOOKUP)GetProcAddress(hMod, MAKEINTRESOURCE(10));
    if (NULL!=pfnLookup)
        {
        pszWord=GetFirstWord();  //Retrieve first word in text.
        while (NULL!=pszWord)
            {
            if (FALSE==(*pfnLookup)(pszWord))  //Check if word is in dictionary.
                [Alert user]
            pszWord=GetNextWord(pszWord);  //Go to next word in text.
            }
        }
    else
        [Export not found, invalid spell checker!]
    FreeLibrary(hMod);
    return;
    }
```

For convenience in terminology, we can call the instance identified with *hMod* a "component" o (the two are not quite equivalent, though often used interchangeably). Describing it as an "obj interesting because what we see here actually meets the core object-oriented principles. The s is *encapsulated* behind the "interface" of **LookUpWord**, because the function is specified outsi implementation. The "object" is also *polymorphic* in that this same client code in AcmeNote wo perfectly fine if we substituted WEBSTERS.DLL in place of BASICSPL.DLL--two implementations specification would be polymorphic. And finally, an instance of BASICSPL.DLL is *reusable*: Som implement another spell-checker, such as MEDDICT.DLL, which only recognizes specific terms field. That DLL could itself load and reuse BASICSPL.DLL in such a way that MEDDICT.DLL chec medical terms, passing all other unrecognized words to BASICSPL.DLL.

Of course, this begs the question of how on earth AcmeNote would know to load MEDDICT.DLL WEBSTERS.DLL instead of BASICSPL.DLL. This brings us to the first in a series of problems tha DLLs are used as a component software model. In the next few sections, we'll look at a numbe detail and the solutions to those problems. These solutions form much of the COM design.

Hard-Coded DLL Names

It should be obvious from the piece of code above that the name "BASICSPL.DLL" is hard-code AcmeNote. This happens either way we write the code: If we call **LoadLibrary**, the hard-codec explicit. If we use an import library, the name is hard-coded in the import record, which we do Such hardcoded names lead to the following problems:

Problem 1

The DLL must exist either in the application's directory, in a system directory, or in some directory included in the PATH environment variable.

Problem 1a

If the DLL is stored in the application directory, it cannot be shared between multip applications that want to use the same service. Therefore multiple copies of the DL end up on the same machine, wasting disk space. (This problem is amplified when there are dozens or hundreds of different DLLs duplicated on a machine. The result might be 10 megabytes of wasted disk space.)

Problem 1b

If the DLL is stored in a system directory, *there can be only one provider of the ser known as BASICSPL.DLL*. Alternatively, each provider must have a different name, in that case AcmeNote will not work if WESBTERS.DLL exists but not BASICSPL.DLL You also run the serious risk that someone else will install a BASICSPL.DLL that is written to a different specification, which will lead to the "Call to undefined dynlink" sort of error messages described earlier.

Problem 1c

If the DLL is stored in some other path, the application must add that directory to t PATH environment variable. On Win32 platforms this is a non-issue, but for Windov 3.1 the size of the PATH is severely restricted.

Much to the delight of hard-drive manufacturers, software vendors usually solve this set problems by installing all DLLs in the same directory as the application. Thus only one ap will ever use that given copy of the DLL. What a waste! Yet the only solutions to these pr must occur outside the DLLs themselves.

Part of the solution is to remove all the path dependencies by defining an abstract identif each DLL. That is, we might define the number 44,980 as being the abstract ID for BASI We then require some way to dynamically map this number to the exact installation poin DLL. This is the purpose of the system registry (or registration database). In the registry create an entry like this:

```
HKEY_CLASSES_ROOT
    ServerIDs
        44980 = c:\libs\44980\basicspl.dll
```

That is, the "ServerIDs" section would list the exact location of many different DLLs. So i using "BASICSPL.DLL" directly in the code above, we would write it like this:

```
void OnToolsSpelling(void)
    {
    wchar_t    szPath[MAX_PATH];
    [Other locals]
    if (!MapIDToPath(44980, szPath, MAX_PATH))  //Looks up ID in registry anc
                                                //returns path.
        [No mapping found, show error)]
    hMod=LoadLibrary(szPath);
    [Other code the same]
    }
```

This at least allows you to install one BASICSPL.DLL in a specific directory so that multipl applications can share it. Each application would only hard-code the ID of 44,980 and no on the exact location of the DLL.

Of course, this doesn't help the same applications use an alternate *implementation* of the *service category*, which is a critical requirement for component software. That is, we'd lik AcmeNote so that it uses whatever implementation of the "Spell Checker" service is arou relying solely on BASICSPL.DLL. Stated in object-oriented terms, we'd like to have all implementations of the same service category (specification) be *polymorphic* with one ar that AcmeNote doesn't care which DLL actually provides the **LookUpWord** function.

(Mind you, a critical requirement for component software is not a requirement for things drivers. Because there's typically only one piece of a certain type of hardware in a machi once, there only needs to be one device driver for that type in the system. For example, have only one video board, you need only one systemwide device driver.)

Certainly AcmeNote would be shipped with BASICSPL.DLL as a default component, but if and better implementation--a WEBSTERS.DLL (ID 56791) with more words and a faster a showed up with some other application, we'd like AcmeNote to automatically benefit.

What we need, then, is an identifier for the *category* along with registry entries that map category ID to the available implementations. So let's say we assign the value 100,587 t spell-checker category. We'd then create registry entries that mapped the category to th implementations, which would be mapped elsewhere to the locations of the modules:

```
HKEY_CLASSES_ROOT
    Categories
        100587 = Spell Checker
            56791 = Webster's Spell Checker
            44980 = Basic Spell Checker
    ServerIDs
        44980 = c:\libs\44980\basicspl.dll
        56791 = c:\libs\56791\websters.dll
```

Now we can write AcmeWord to use the most recently installed spell-checker (which coul inferior one), or we can add a user interface such as that shown in Figure 2, which allows to select the preferred spell-checker from those listed in the registry. We'd then have coo follows:

```
void OnToolsSpelling(void)
    {
    DWORD       dwIDServer;
    wchar_t     szPath[MAX_PATH];
    [Other locals]
    dwID=MapCategoryIDToServerID(100587);  //Magic to get a server ID, may
                                           //involve UI
    if (!MapIDToPath(dwID, szPath, MAX_PATH))  //Looks up ID in registry and
                                               //returns path
        [No mapping found, show error)]
    [Other code the same]
    }
```

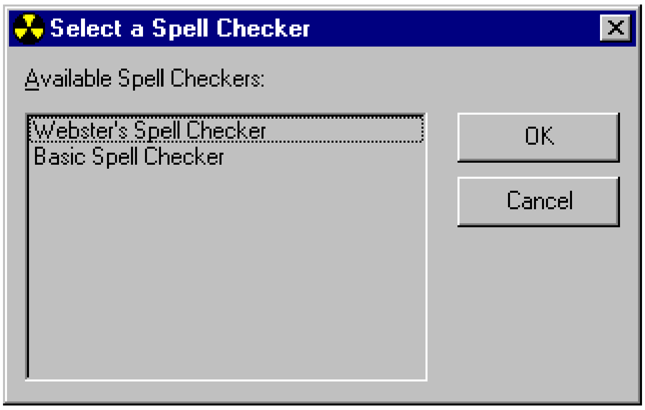


Figure 2. A hypothetical selection dialog based on registry entries

So with category identifiers, server identifiers, and a registry we can solve Problem 1. CC OLE use the registry in much the same way to bypass exactly these problems. For examp Documents," the OLE specification for compound documents, describes what are categori "insertable" (that is, embeddable) content objects. The various implementations of such are described in the registry, and when users want to insert a piece of content into a doc they invoke the standard "Insert Object" dialog, which lists the available object types mu dialog shown in Figure 2. OLE Controls, defined in another specification, operate in the sa manner.

However, the use of abstract identifiers presents another problem itself:

Problem 2

Who defines the category IDs and the server IDs?

There are two possible solutions for this:

- One, some central organization takes responsibility for maintaining a master list of assigned to categories and IDs assigned to DLLs. This means that vendor wishing t define a category or ship an implementation of some category must get "permissio by obtaining unique identifiers from the organization controlling the master list. Ble
- Microsoft realized that a centralized approach like the one described above would s innovation and be a royal pain to manage. Therefore COM and OLE use *globally uni identifiers* (GUIDs, pronounced goo-ids or gwids)--128-bit values that are generate algorithm defined by the Open Systems Foundation to guarantee uniqueness acros space. Microsoft provides a tool that implements the algorithm so that anyone, any anytime, can obtain identifiers as needed and still be assured uniqueness.

Of course, it is also wasteful for each application to define its own category IDs and its ov structure, so Microsoft has defined some standards in these areas. COM and OLE provide manipulate GUIDs and their various uses in the registry. We'll see more details later in tl For now, simply understand that *COM eliminates dependencies on DLL names, thus allow multiple polymorphic providers of the same service to coexist.*

Management APIs

You may have noticed that the code in the previous section contained two undefined "ma functions: **MapCategoryIDToServerID** and **MapServerIDToPath**. The first function lo ID of a server for a particular category that may display a UI similar to Figure 2. The sec function retrieves the path of the server DLL associated with a particular ID. We might ha combined all these steps, including the call to **LoadLibrary**, into one function named sor like **LoadCategoryServer**.

Such functions are an example of a "management API," which frees an application from t of walking through the registry and so forth. But the API example shown here is actually advanced one because we have a registry and have defined some kind of standards for tl structure of the registry. Therefore the API can be somewhat generic and accommodate different service types.

However, this was not always the case. For a long time there was no central registry and were no standards for using any other means to store such information. The result was t whenever someone defined a new service category, they also defined a category-specific management API. The intention was, of course, to make it easier to write an application services in that category, and indeed such APIs did for their respective categories. In our we might have a specific **LoadSpellChecker** API function that essentially hard-coded the ID. On the surface, this does simplify the programming model in our code.

Over time, many new categories were defined, each with their own management API. In the API initially made working with an individual service type easier, but overall they con application programming. Because most applications employ many different services to a their desired ends, having a specific API for each service meant a steep learning curve fo programmers involved. The Win32 API itself, evolved over a decade, is a prime example: approximately 70 different "creation" functions, each dealing with a different kind of "obj as **CreateAcceleratorTable**, **CreateDialog**, **CreateEvent**, **CreateMailslot**, **CreateMu** **CreateRectRgn**, and **CreateWindowEx**. Each type of "object" also has a different way are hundreds of API functions, each working in different ways, which is what makes prog for Windows so difficult.

We may state the problem as follows:

Problem 3

> New service categories introduce new management and manipulation APIs, meanin simplify the programming model for that category, but tend to complicate the over programming of an application.

The solution is an obvious one: Create generic abstractions for the common elements fou these APIs. This is what the **MapCategoryIDToServerID** and **MapServerIDToPath** fu doing in our example. The process of finding the existing implementations of a category i common one in nearly all category definitions. By defining a generic API based on unique identifiers, we greatly reduce the overall number of API functions and thereby truly simp overall programming model.

This is exactly what COM does--*provide a single generic API for the management of all ca and servers*, based on the standards defined for how these things are listed in the registr generic API is extensible in that it can accommodate any new categories and services un of time. Furthermore, this API is made up of only a handful of functions, which can be lea matter or hours or days, the most important of which is named **CoCreateInstance**. This creates an instance of some class (given a CLSID) and returns an interface pointer for it.

A key to this generic API is that object instances are always referenced through an "inter pointer," the exact structure of which we'll see later. For now, suffice it to say that all int pointers are also polymorphic in a way that allows a generic API to manipulate any kind c interface pointer to any kind of object. This stands in stark contrast to the myriad non-pc handles and references that one finds in the Win32 API.

Shared Server Instances and Lifetime Management

The last section mentioned that many different service types in the Win32 API have diffe means to identify an "instance" or "object" of the service, component, or object of concer code we've looked at so far, the spell-checker "object" is identified with a module handle which you can get at function pointers. This is all well and good for the application that lo DLL, because the module handle makes sense in that application's process space. Howev same handle is useless to another process. That is, our AcmeNote application could not p module handle to another application (via remote procedure call--RPC--or some other int communication mechanism) and expect that other process to make any sense of it. Henc another problem (or limitation, one might say):

Problem 4

> In general, identifiers for instances or for a server or component cannot be shared other processes, let alone with processes running on other machines.

On Win32 platforms, process separation means that each application wanting to use a ha based service must load that service into its own address space, wasting memory (even ' memory), increasing boot time, and causing an overall performance degradation. The on workaround for this on Windows NT is to create a subsystem type of service where only c instance is running for the whole system. Much of Windows itself (like USER.EXE) works which is why certain handles, like HWNDs, can be passed between processes. But these t system services are expensive and are not suitable to the general problem of component *across machines*--that is, have multiple applications on multiple machines accessing a se that's running on yet another machine, like a central database. In order to make distribu systems work you must go to full-blown RPC, or named pipes, or another such mechanis which of course have structures and programming models different from anything you m on a single machine or within a single process. Thus we have another problem:

Problem 5

> The mechanisms for working with services differ greatly between the in-process (D local (.EXEs, other processes on the same machine), and remote (other machine) cases. A client application ends up having to understand three or more different programming models to work with all three service locations.

With DLLs in particular, there are other problems that would arise if you *could* share moc handles between processes. Let's say AcmeNote has loaded BASICSPL.DLL and then pass module handle to another application. What happens when AcmeNote terminates and the application still has the module handle? Because the DLL is loaded into AcmeNote's proce DLL is unloaded as well. Thus the module handle becomes invalid and we can expect the application to crash the next time it attempts to use that handle.

There's a similar problem on 16-bit platforms like Windows 3.1, where all applications ru same address space and one instance of a DLL could be used by multiple applications. Th there is that abnormal termination of one application might cause the DLL to remain in m even when all other applications using it are closed normally. Programmers for Windows usually become proficient with a little tool called WPS.EXE, which allows you to clean orp DLLs out of memory.

In both cases we can state yet another problem:

Problem 6

> When instances of a service can be shared between processes, there is usually no robust means of handling abnormal termination of one of the processes, especially one that first loaded the service. Either the server remains orphaned in memory or other processes using that service may crash. DLLs, for example, have no way of knowing when other processes are referencing them.

COM solves all of these problems by virtue of the "interface pointer" structure (introduce previous section) as well as by providing the ability for you to implement servers as *eithe* .EXEs. Through a process called "marshalling," one can share an interface pointer with ot processes on the same machine or on other machines. Of course, you don't pass the exa pointer value, but the result is that multiple processes can jointly own an interface pointe which they can call functions in the object attached to that pointer. The structure that m possible is borrowed from RPC and also guarantees robustness when the process that loa object is terminated. When other processes try to use that object, they receive "disconne error codes in return, instead of just crashing. Also, if an object is implemented in its ow has its own process, it will remain active as long as any client holds a reference to it. (CC automatically releases references held by clients that terminate abnormally, so the serve doesn't get orphaned itself.)

What's more, the *only* way one ever deals with an object instance in COM and OLE is thr interface pointers. This applies regardless of the actual location of the object itself. There programming model is identical for the in-process, local, and remote cases. The technolo makes this work is called "Local/Remote Transparency," as we'll see later.

Multiple Services in a Single DLL

Early in this section we described a *server* and a *component* as pretty much the same th we used the terms *component* and *object* to describe a loaded instance of a server. We c this because of the base assumption that each server DLL only implements a single servi

However, anyone who has been doing Windows programming for very long knows that a implies a fixed amount of overhead, both in memory footprint and in the time it takes to DLL. When performance is a big issue, the first thing you'll want to do as a component ve combine services into one DLL to improve the boot time and working set of applications t your DLL.

The problem here is that when services are defined as a set of DLL exports, allowing one offer multiple services requires some kind of central coordination as to which ordinal num and/or function names are assigned to which service categories. That is, if two categories want to use ordinal 10 (or the same text name) for completely different operations, you implement both services in one DLL! So once again we're right back to the issue of havin central body (that is, bottleneck) having to grant "permission" to those creating new des

Previously we saw how COM and OLE use GUIDs to identify both service categories and implementations of a category, which allows anyone to independently define or implemen service without having to talk to anyone else. In order to allow multiple services per DLL to solve the following problem:

Problem 7

> A server cannot support multiple services without risking a conflict between the function names or ordinals used to reference those functions. The choice seems to between central control over ordinal assignments and the inability to combine serv together, either stifling innovation or causing performance problems.

The real solution is to get away from referencing functions with names and ordinals altog Earlier we saw that the absolute identity of any given function in a DLL-based system is **module:ordinal**, for example, **BASICSPL:10**. In COM and OLE, components and objects their functions not through singular exports, but in groups that can be referenced throug interface pointer. An "interface" is actually defined as a group of related "member functio that group is itself assigned a GUID, called an Interface Identifier, or IID, which anyone generate independently like all other GUIDs.

For example, if we were to redesign the spell-checker specification to use COM, we'd defi interface called **ISpellChecker** as part of the specification (the conventional "I" prefix to stands for "interface"), as follows:

> A spell checker component is any server supplying an object that implements the ISpellChecker interface which is defined as follows using the Interface Definition Language (IDL):

```
/*
 * IUnknown is the "base" interface for all other interfaces. The value
 * the uuid() attribute is the IID, which is given the symbol IID_ISpel
 * ISpellChecker is the interface type as recognized by compilers.
 */
[uuid(388a05f0-626d-11cf-a231-00aa003d7352), object]
interface ISpellChecker : IUnknown
    {
    HRESULT LookUpWord(OLESTR *pszWord);
    }
```

Notice how we've eliminated any reference to ordinals as well as any need to stipulate th

spell checker must be implemented in a DLL. Implementing a server according to this sp
is hardly any more work than implementing the DLL according to the old specification.

As we'll see in detail later, an interface pointer (a run-time entity) really points to a table function pointers. In the case of **ISpellChecker**, the table has four entries--three for the of the base interface **IUnknown** and one for **LookUpWord**. The table contains the addr the implementations of these interface member functions. When a client wants to call a function through an interface pointer, it actually calls whatever address is at the appropr in the table. Programming languages make this easy, given the way interfaces are define header files. AcmeNote can now use COM's generic **CoCreateInstance** function and the to meet this new specification (this is C++ code), as follows:

```
void OnToolsSpelling(void)
    {
    ISpellChecker *pSC;
    CLSID          clsID;
    //This hypothetical function uses the registry or invokes the UI.
    GetCLSIDOfSpellChecker(&clsID);
    if (SUCCEEDED(CoCreateInstance(clsID, NULL, CLSCTX_SERVER
        , IID_ISpellChecker, (void **)&pSC)))
        [Spell-checker not found, show error]
    pszWord=GetFirstWord();  //Retrieve first word in text.
    while (NULL!=pszWord)
        {
        if (S_OK!=pSC->LookUpWord(pszWord))  //Check if word is in dictionary
            [Alert user]
        pszWord=GetNextWord(pszWord);  //Go to next word in text
        }
    pSC->Release();
    return;
    }
```

Given the interface definition, a compiler will know to generate the right machine code th the right offset for the **LookUpWord** member of **ISpellChecker**. To do this reliably requ standard for the binary structure of an interface--one of the fundamental parts of the CO specification.

Now, putting all this together, we see that any given function--in any given object implementation--is identified with three elements: the object's CLSID, the interface type through which the client will call member functions, and the offset of the particular mem function in the interface. Thus, the absolute function identity in COM and OLE is **CLSID:IID:table_offset**; for example, **CLSID_BasicSpellChecker:IID_ISpellChecke**

Because anyone can generate CLSIDs and IIDs at will, anyone can design a service wher functions describing that service are ultimately identified with two absolutely unique valu

COM allows any given server module to implement as many different classes as it wants, that any server can support as many CLSIDs as desired. When a client wants to access a of a class, COM passes the CLSID to the server and the server can decide what to instant which interface pointer to return. That interface pointer points to a table that holds the a of the code that is unique to the particular object class. In short, COM removes all barrie multi-service implementations, regardless of who designed the service.

A small note about the **pSC->Release()** line at the bottom of the code above: **Release** member of the **IUnknown** interface that all interfaces share, thus all objects have this n *Release* is how a client tells the object that it is no longer needed. The object maintains a count for all clients using it, so it knows when to free itself from memory. In addition, a s maintain a count of objects it happens to be servicing, so that when no objects remain it unload itself (or terminate, if it's an .EXE server).

are only a handful of other special-case "free" or "delete" functions in the whole OLE API. all objects are manipulated through interface pointers, freeing the object always means c **Release** member. This single yet powerful abstraction significantly simplifies the overall programming model, just like the generic creation function **CoCreateInstance**. In fact, most common patterns in COM/OLE programming is: *(1) Use **CoCreateInstance** to obta interface pointer, (2) call functions through that pointer, (3) call **Release** through that p when the object is no longer needed*. You see it everywhere, sometimes involving a diffe creation function, but you still see the pattern.

The Big One: Versioning

We've worked our way now through many of the problems inherent in DLL-based service and we've introduced many of the solutions found in COM and OLE. But everything we've with so far applies only with the *first* version of a particular service. Not only the first ver category definition, but also the first version of the server implementation, and the first the hypothetical AcmeNote application that uses such a server.

The big question is what happens when we want to (1) change the service specification, the server, and (3) update the client. You *will* eventually want to do this because innovat the heart of the software industry! The primary issue becomes one of compatibility betw different versions of the services and the applications that use those services. A new ser be prepared to work with an application that expects an old service. A new application m prepared to work with an old service.

This is called the "versioning problem," which happens to be the most important problem was designed to solve because it is one that has historically made many lives miserable. applies even to those services that only require one provider in the system! The problem

Problem 8a

> Versioning: Independently changing the specification of a service, the implementat of a server, or the expectations of a client typically results in significant interoperal problems that ultimately prevent innovation and improvements in software.

Or, stated another way:

Problem 8b

> When an application talks directly to a service, how does that application dynamica and robustly discover the capabilities supported in that service? How does an application differentiate between different versions of the same service category ar between different versions of a server implementation?

It is quite a claim to say that COM fundamentally solves this problem across the board, y true. We must first, however, understand the problem, which we'll do by going back to tl specification for the spell-checker DLL. The specification stipulated that a component, wh call "SpellCheck1.0," is a DLL that exports a function called **LookUpWord** as ordinal 10.

Now we've happily created BASICSPL.DLL, which we'll call "BasicSpell1.0" for convenienc DLL is used by AcmeNote version 1.0. At this stage, interoperability is a non-issue as lon everyone sticks to the API specifications--this is *always true* with version 1.0 of *any* API. sticking to the specifications, one is free to make performance improvements that don't a interface between client and service. So BASICSPL.DLL can ship a new version with more and a faster lookup algorithm to compete with WEBSTERS.DLL. No problem.

Problems arise when we want to change the *specification*, usually because we want to in example, customers will soon want the ability to add custom words to the dictionary. Thi require new functionality in spell-checker components. Working in the DLL world, we wou the SpellCheck2.0 specifications like those shown in Table 1.

Table 1. A Spell-Checker 2.0 Component (DLL) That Exports Functions

Ordinal	Prototype	Description
10	BOOL WINAPI LookUpWord(wchar_t *pszWord)	Checks to see if a particular word exists current dictionary.
11	BOOL WINAPI AddWord(wchar_t *pszWord)	Adds a custom word to the dictionary.
12	BOOL WINAPI RemoveWord (wchar_t *pszWord)	Removes a custom word from the dictior

Note that a SpellCheck2.0 implementation is polymorphic with a SpellCheck1.0 implemer because 1.0 clients will only look for ordinal 10. It seems that we could overwrite the DLl BasicSpell1.0 with a DLL for BasicSpell2.0, but as we'll soon see, there are many potenti problems.

Earlier we saw two different versions of AcmeNote code that used the **LookUpWord** func using an import library to resolve the function name, the other retrieving the address of function directly with **LoadLibrary** and **GetProcAddress**. Both solutions work equally w there's only one function to worry about. But when we want to write AcmeNote2.0 and a "Add/Remove Word" feature, the additional functions make it more complicated to progra **GetProcAddress** than to use import libraries. It's still tolerable for AcmeNote2.0 becaus functions are used at the same time and we only need one call to **LoadLibrary**.

But any worthwhile application will not use just one service DLL, nor only a handful of fu the same place in the code. Most likely there will be dozens of functions strewn all aroun application. Calling **LoadLibrary** and **GetProcAddress** for each call introduces many mc conditions, which rapidly increases the complexity of the application and thus the bug co real world, vendors are concerned with shipping the application in order to make money, more robust means of dynamic linking quickly gives way to the use of import libraries. A having implicit links like this isn't so bad, is it?

Well, let's say we ship AcmeNote2.0 along with BasicSpell2.0, which happens to be in the module as BasicSpell1.0 (that is, BASICSPL.DLL). When we install AcmeNote2.0, we over old BASICSPL.DLL with the new one. We know that any application that was using BasicS will still be able to use BasicSpell2.0, so compatibility is ensured.

However, there now exist two different versions of BASICSPL.DLL that, to most people, a indistinguishable from one another. Microsoft has frequently discovered that older versio DLL end up overwriting newer versions, no matter what rules you lay down. Some applic uses BasicSpell1.0 might come along and overwrite the newer BASICSPL.DLL with the ol (The reason for this is that it is difficult to do the right thing--to check the existing DLL v against the one you're installing. This is tedious and it's normally very hard to find a prog who wants to work on an application's installation program. When crunch mode comes a want to ship an application, you're much more likely to spend time fixing application bug making your installation program perfect. Checking version resources on shared libraries thing to compromise.)

So what happens when the user now runs AcmeNote2.0? When that application tries to r **AddWord** and **RemoveWord** calls, it will fail. If import libraries are being used, which is common, the kernel will generate its enigmatic error messages like the ones we saw bef customer support calls to come streaming in.

No matter how hard one tries to prevent this situation, it invariably happens when multip versions of the same service use the same DLL name. There are several ways to prevent problem that are less than optimal: punt, bloat, stagnate, or suffer. Let's look at these in before we see how COM solves the problem for good by making the "right" thing to do ar part of the whole programming model.

Punting: Programming to the Least Common Denominator

The easiest way to avoid "Call to undefined dynlink" problems is simply to avoid using a version of a DLL in the first place! This is often an appealing solution because it takes no carries no risk, but it means no innovation.

The result is called the "least common denominator" usage of the feature set of a service meaning that almost no one uses any functionality that is not guaranteed to be present i *versions* of that service DLL. What happens then is that the vendor of the DLL is hesitant any more features to the DLL because few applications will bother using them--there is li on investment. Who is going to take the risk first? The DLL vendor or the application ven commonly, neither.

Usually the only way to avoid this sort of trouble is never to allow the service DLLs to be distributed by themselves, thereby avoiding the possibility that an old DLL would overwri DLL. This basically means that the new service DLLs must be shipped with a new version operating system, and that application should be revised and recompiled to work on the operating system. Anyone who has been around the Windows development scene for a fe knows how this works: As Microsoft prepares a new version of Windows, it heavily evang independent software vendors (ISVs) to produce new versions of applications that take a of the new features in the system, thereby making the old system obsolete, thus encoura users to upgrade the operating system, the applications, the hardware, and so on.

Microsoft must do this because otherwise the new system will be a complete flop. Micros must invest tremendous resources in making sure that old applications still do work, bec without that backwards compatibility the system is also doomed. Not only are evangelism compatibility testing expensive, but the system ends up bigger and slower because of it. course, this doesn't even start to solve the problem for vendors who want to ship their o services!

Bloating: Shipping Multiple DLLs with Redundant Code

Another solution that avoids both the least-common-denominator problem and the versio problem is always to ship new features in an altogether new DLL. This happens in one of Either you add new code to the existing DLL code base and recompile the whole mess, or a new DLL that contains only the new features. Each solution is basically saying, "DLLs n versioned at all!"

In the first case, the DLLs themselves are given a filename that reflects the version num as BASSPL10.DLL, BASSPL20.DLL, etc. Therefore all the versions of the server DLL can c there is no chance of overwriting the old version because it has a different name.

This is, in fact, how Microsoft has traditionally solved the versioning problem, as with the Basic run-time libraries in VBRUN100.DLL, VBRUN200.DLL, VBRUN300.DLL, and so on. Th however, is that these things build up on the user's machine. On the machines I'm worki right now, I have the following versions of the 16-bit Visual Basic run-time library: VBRU (348K), VBRUN300.DLL (389K), and VB40016.DLL (16-bit VB 4.0 library, 913K). In all, 1 Visual Basic run-time libraries!

my question is this: How much code is repeated between these DLLs? I would bet that th
fair portion of code that exists identically in all of these DLLs. Ideally I should need only t
recent DLL to work with all versions of the applications that use the DLL's services. On m
machine, therefore, I should only need the 913K of VB40016.DLL and I should be able to
of old stuff. Now by itself this isn't bad, but multiply the problem over time with a few ad
versions and by dozens of DLLs. Sooner or later I might have 50MB of disk space being e
worthless DLLs that never get used. Yet I'm afraid to delete them in case one of my appl
needs them!

Unless you're a hard-drive manufacturer, this sucks. Users would love to know when the
delete some of these old DLLs to make more disk space, and there are software products
market that try to help users decide whether or not some old DLL is in use. (Occasionally
some old DLLs to a "DeleteMe" floppy and work on my machine for a while--if nothing us
DLLs I'll eventually delete them.)

So why does Microsoft put redundant code into each version of these DLLs? Why not just
VBRUN200.DLL only what is different from VBRUN100.DLL, and do the same for subsequ
versions?

Well, there are two reasons. One is that this still bloats the user's machine with a bunch
albeit smaller, but still a lot of clutter. The other reason--the big one--is performance. As
mentioned before, each DLL has a fixed overhead in load time and memory footprint. Inc
the number of DLLs can exponentially increase application load time. In an industry that
applications partially on their boot time, this simply is not acceptable.

So in order to sell its applications, Microsoft and all other vendors usually decide to shift
to the user, who now must buy more disk space or more powerful machines. Microsoft do
having to do this, but when it's a tradeoff between selling the application and having the
spend more on hardware, or not selling the application at all, the former is chosen hands
course, the hardware manufacturers don't mind one bit.

Stagnating: The Service-Level Approach

Another possible solution is to involve a specific management API that allows the applica
negotiate with a server to determine if that server has support for a particular "level" of s
We've already seen that having a specific management API is a bad idea; still, this soluti
been used in the past.

The idea here is that instead of having just one "functional API" exposed from the server
different API "levels," each level consisting of a group of functions that together represen
additional features. New features are introduced in new levels. Thus the "Level 2 API" ex
base API, "Level 3" extends "Level 2," and so on. The provider of a Level *n* service is poly
with all other <*n* providers.

Each application that wants to use a service through this API asks the management API t
those services that implement a certain level of API, using registry entries. Thus the app
will guarantee that it will be working with a server that implements the right functionality
it can disable certain features in order to work with a down-level server.

The application doesn't actually link to the functions exported from the server itself beca
doesn't know the DLL name ahead of time; rather, it links to a "stub" server that provide
points for all functions in all levels. When the application asks the management API to lo
server, that server is loaded and dynamically linked into the stub server (using **GetProcA**
Client calls into the stub server are then forwarded to the real server.

A key feature in this kind of architecture is the grouping together of functions as "levels."
level is a "contract," which means that when a server implements one function in a level
implement *all* functions in that level. This enables the client application, having successfu

will work as expected. The client only has to check for Level 3 support once before it can any function in Level 3.

This "contract" idea also means that the application doesn't have to ask the provider abo function call before invoking that function. Without the manager API, determining "levels service in some random DLL would mean calling **GetProcAddress** for each function befo to call that function. The next section examines the implications of this method.

So why isn't this "level" architecture the right solution? Because a number of the problen discussed earlier are still present in this model:

- First of all, who defines each level? Because a level is defined as a group of exporte functions, we still have the problems of centralized control over the specification.
- Second, we have a specific management API with all its inherent problems, includir that it is a tedious process to define such an API, which must be done at each level Solutions like this end up being designed in a committee, which is a notoriously *slo* innovate; the result is usually stagnation.
- The other big problem is that this *really doesn't solve the versioning problem*, beca architecture doesn't make any provision for changing the definition of a *level* itself. way to add new functionality is to define a new level. But what if you have a new fe belongs in Level 2 but you have to add it as Level 6? That means that to implemen feature you must implement Levels 3, 4, and 5, even when you hadn't before. Unle design each level perfectly from the beginning, the "level" approach here just does complicates everything while solving almost nothing.

Suffering: Programming with GetProcAddress

The fourth way around the "call to undefined dynlink" problem is to return to the **GetPrc** programming model to avoid using import libraries altogether. Although this is a difficult programming model to work with, it may be the only choice for an application that really robust. Again, the complication is that every call into a DLL turns into two or three calls: **LoadLibrary**, **GetProcAddress**, and the actual call desired. At a minimum, complexity v and in our competitive industry this can kill a product (or even a company).

Developers have invented all kinds of interesting strategies to retain the robustness of th programming model while reducing the complexity. One quick optimization is to load any only once. This means that the application has one global HINSTANCE variable for each D wishes to load in this manner, and each variable is initialized to NULL. Whenever the app needs a certain DLL, it calls an internal function to retrieve the module handle. This func checks if the global variable is NULL, and if so, loads the DLL; otherwise it simply returns handle that already exists in the global variable. For example, we might have a global ar module handles and symbols for the array index of each DLL we're going to use. Our inte helper function checks the array and loads the module if necessary, as shown here:

```
#define CMODULES           10  //Number of DLLs
#define DLL_ID_SPELLCHECK  0   //Array indices
#define DLL_ID_THESAURUS   1
[other defines here]
typedef BOOL (WINAPI *PFNLOOKUP)(wchar_t *);
HINSTANCE g_rghMods[CMODULES];  //Initialized to NULL on startup.
HINSTANCE ModHandle(UINT id)
    {
    ASSERT(id >=0 && id < CMODULES);
    if (NULL==g_rghMods[id])
        {
        g_rghMods[id]=LoadLibrary([name of DLL]);
        if (NULL==g_rghMods[id])
            [Throw exception]
        }
    return g_rghMods[id];
```

```
    }
void OnToolsSpelling(void)
    {
    PFNLOOKUP  pfnLookup;
    [Other locals]
    pfnLookup=(PFNLOOKUP)GetProcAddress(ModHandle(DLL_ID_SPELLCHECK)
        , MAKEINTRESOURCE(10));
    if (NULL!=pfnLookup)
        {
        [do stuff]
        }
    else
        [error]
    return;
    }
```

The array in **g_rghMods** would be initialized to NULL on startup, and on shutdown the a would call **FreeLibrary** on anything in here that was non-NULL. (Or this might happen w freeing up memory.) Throwing an exception on **LoadLibrary** failure would allow us to wr kind of code shown in **OnToolsSpelling** where we don't have to check for failure of **Mod** we put that code in an exception handler. All of this takes us quite far towards simplifyin programming model throughout the entire application.

Another complication is that the pointers returned from **GetProcAddress** are defined as FARPROC, or as a pointer to a function that takes no arguments and has no return value. you define specific types for each function pointer you're going to use, such as PFNLOOK code above, you'll get no compile-time type-checking. This means that enormously critic time errors can arise when the application either calls a function without the requisite ar or misinterprets a return value. Even if you take the time to write all the necessary type probably mistype some of them or use the wrong type somewhere in the source code. Yo have run-time errors, now caused by extremely hard-to-find bugs in header files and sou otherwise compile without warnings!

The right answer to this problem is for server vendors to provide you with such typedefs header files to begin with. Ultimately the types should be defined by those implementing service, instead of placing the burden on the application. As we'll see, COM enforces this stronger way that allows strong type checking while eliminating the possibility of bugs ca subtle definition or usage errors.

Now that we have workable solutions for the loading and typing issues, how can we simp obtaining the function pointers themselves? It complicates the programming model to ca **GetProcAddress** each time we need a pointer. A more efficient way to do this would be all the pointers we might need when we first load the DLL, using the same technique as with the module handles. That is, for each DLL we define an array of function pointers. W change our **ModHandle** function to be **FunctionPointer** so we can make calls like this:

```
void OnToolsSpelling(void)
    {
    [other code]
    while ([loop on words]
        {
        if (!(*(PFNLOOKUP)(FunctionPointer(DLL_ID_SPELLCHECK
            , FUNC_ID_LOOKUP)))(pszWord))
            [etc.]
        }
    return;
    }
```

Again, we can use exceptions so that **OnToolsSpelling** doesn't have to check for a NULL value from **FunctionPointer**. We might also write macros for each function we wish to u don't have to write such ugly code. In the end, we can be writing code like this:

```
#define CALL_SPELLCHECK_LOOKUP(p)  (*(PFNLOOKUP)(FunctionPointer(DLL_ID_SPELI
    , FUNC_ID_LOOKUP)))(p))
void OnToolsSpelling(void)
    {
    [other code]
    while ([loop on words]
        {
        if (!CALL_SPELLCHECK_LOOKUP(pszWord))
            [etc.]
        }
    return;
    }
```

Now we've simplified the programming model to the point where it is just as easy to wor when we were using import libraries, yet we avoid all the import library trouble.

But what happens when we have multiple versions of the server to deal with? We have t that some of the functions we want to put in our table of pointers will not actually exist i server. How should we handle this? A sophisticated application would do something like t

- Define multiple function tables for each server, where each table represents a grou functions that make up a certain "feature." The application itself would choose the groupings. For example, there might be one table for "basic spell checking" that co only **LookUpWord**, one for "dictionary additions" that contained only **AddWord**, o "dictionary editing" that contained both **AddWord** and **RemoveWord**.
- Define a flag for each "feature" that is set to TRUE only if all the necessary functior available for that feature.
- Using these flags, allow other application code to enable or disable certain user cor depending on which features were available from the various servers. For example, checking was available, the Tools/Spelling command would be enabled. The spell-c dialog might have Add and Edit buttons inside it, where Add would be enabled only "dictionary additions" feature were available, and Edit would be enabled only if the "dictionary editing" *and* "dictionary addition" features were available.

In addition, a *really* sophisticated application would provide its own default code for certa functions that might not exist in all servers, rather than disable the feature altogether. F example, a sophisticated word processor might provide its own backup custom dictionary implementation if a suitable server weren't available. When the application discovered th of **AddWord** and **RemoveWord** functions in the server, it would store its own entry poir table instead. Of course, the application would then install its own proxy implementation **LookUpWord** to filter out custom entries before calling the server's implementation.

Alternatively, the application might store a pointer to a "do-nothing" function in the table the rest of its code could trust that the table is completely full of valid pointers, even if s functions don't do anything. That's better than having to check for NULL entries in the ta

In short, a really sophisticated application would define and guarantee its own idea of a " for certain features without complicating its own internal programming model.

Sounds great, doesn't it? Assuming that we can solve all the other problems with the reg generic management APIs, and so on, the reality is still that (1) it is *very costly* to create application architecture like this, and (2) such an architecture creates a bloated applicati might contain 30 to 50 percent more code than one that just lived with import libraries. : when robustness is the top priority and resources are not an issue, you might do this. Bu honest, 95 percent of all vendors probably cannot afford going to this extent.

But what an awful burden to place on every application! Why should applications have to burden on every application for just one DLL, let alone dozens that they might use! Is it appropriate to place the definition of "features" on the servers themselves? Is it not appr have the servers provide the do-nothing stub functions? Should not the servers create th themselves?

> The answer is YES! They should! It is wholly appropriate to have only one implementation of this table-construction code (and instances of the tables themselves!) inside the server rather than have it duplicated across many applicati thus causing code bloat (and happy hardware manufacturers!). It is the *server* that should be responsible for fulfilling the "contract" defined for a certain feature, not t *client*! The client should be able to just ask the server, "Do you support this contra and if the answer is yes, invoke any function in that contract without having to che for the existence of individual functions, without having to create tables, without having to provide default implementations, and--most important--without having t any of the work to define the function types, the macros, and the helper functions.

We are now ready to see that the COM architecture of objects and interfaces, especially *interfaces*, gives us *exactly* what we need to solve versioning without undue burden on c without unnecessary complexity for servers.

The Design of the Component Object Model

In the preceding sections I've hinted at how the Component Object Model (a combination of sp standard system-provided implementation) provides the solutions to a wide range of requireme component software architecture. Table 2 below reviews what we have already learned.

Table 2. Summary of Solutions to Component Software Problems

Problem	**Solution**
Path dependencies, multiple providers of a service	Use the registry to map from abstract class identifiers to absolute server locations as well as to map between categorie and class identifiers.
Decentralized definition of identifiers	Use GUIDs generated by an algorithm to guarantee uniqueness across time and space, eliminating the need for centralized identifier allocation.
Specific management APIs for each service category	Supply a very simple generic and universal management API that accommodates all service categories, called "Implementation Location"
Sharing object instances across process/machine boundaries	Implement marshalling of interface pointers with Local/Remot Transparency providing the ability to implement a server as a .EXE or .DLL..
Different in-process, local, and remote programming models	Design a single model for all types of client-object connections supported through the Local/Remote Transparency interface structure.
Lifetime management of servers and objects	Institute universal reference counting through the **IUnknown** base interface that all objects support and from which all othe interfaces are derived.
Multiple services per server module	Use the **CLSID:IID:table_offset** syntax instead of **module:ordinal** to absolutely identify functions.
Versioning	Support the concept of multiple immutable interfaces as well as interfaces that are strongly typed at both compile time and

	run time.

The following sections describe GUIDs, interfaces, implementation location and the registry, lo transparency, the **IUnknown** interface, and how all of these elements help solve the versionin

Globally Unique Identifiers

Within the scope of an application or project where one group controls all the source code, it is naming conflicts between functions, objects, and so on. The compiler will tell you if there's a co component software environment, however, especially a distributed one, you have the possibil different developers in different organizations having to name functions and modules and class running into conflicts. This can either be coordinated through a centralized organization, which and expensive, or it can be achieved by a standardized algorithm such as the one created by tl Systems Foundation (OSF) for their Distributed Computing Environment (DCE).

In this environment, remote procedure calls (RPC) need to travel around large distributed netw exactly what piece of code must be called. The solution is an identifier standard called the Univ Identifier (UUID) generated with an algorithm that uses a machine's Internet protocol (IP) add *space*, guaranteed unique by network hardware manufacturers) and the current date/time whe is run (unique in *time*). Uniqueness in both time and space means that no matter when and wl algorithm is run, it will produce a unique value. This is possible because UUIDs are 16-byte val in which you can create a staggering 3.4x10³⁸ combinations. Put another way, i the world (5.5x10⁹) generated ten trillion (10¹³) GUIDs per secor clock, it would take *196 million years* to run out of GUIDs. I desperately hope that all this softv obsolete by that time!

COM and OLE use UUIDs for all unique identifications needs, simply calling them "globally uniq or GUIDs, instead. (Claiming *universal* uniqueness is a bit pretentious--other civilizations on ot be using the same IP addresses. So COM calls them *globally* unique, which is more accurate!) DCE standard, GUIDs and UUIDs are spelled out in hexadecimal digits in the following format: 1234-1234-1234-012345678AB}.

The algorithm itself is commonly implemented in a tool called UUIDGEN, which will spit out one any time you ask. The Win32® SDK tool supports a command-line switch, *-nXXXX*, to generate sequential UUIDs. Another Win32 SDK tool, GUIDGEN, generates one GUID at a time in a varie formats that are more directly useful in source code.

You use these tools whenever you need a GUID to assign to a category specification, a class im or an interface definition, trusting the absolute uniqueness. And because interfaces are the fun extension mechanism in COM and OLE, you can innovate without having to ask permission fron

Interfaces

Interfaces are the point of contact between a client and an object--it is only through an interfa and object communicate. In general, an interface is a semantically related group of member fu carry no implementation (nor any data members). The group of functions acts as a single entit essentially represents a "feature" or a "design pattern." Objects then expose their features thro more interfaces, as we'll see shortly.

The definition of an interface is that of an "abstract base class" in C++ parlance. Interfaces are described using the Microsoft COM extensions to the standard DCE Interface Definition Langua seen one example already for a hypothetical **ISpellChecker**, where the "I" is a conventional p "interface" (the **object** attribute is the IDL extension that describes a COM interface as oppose interface):

```
[uuid(388a05f0-626d-11cf-a231-00aa003d7352), object]
interface ISpellChecker : IUnknown
    {
    HRESULT LookUpWord(OLESTR *pszWord);
```

}

The Microsoft IDL (MIDL) compiler can generate header files and various other files from an ID
At compile time, the interface name (**ISpellChecker** in the example above) is a data type that
type checking. The other important element is the **uuid** attribute, which assigns an IID to the
IID is used as the run-time type of the interface. In other words, the IID identifies the intent o
design itself, which means that once defined and assigned an IID, an interface is immutable--a
requires assignment of a new IID because the original intent has now changed.

At run time, an "interface" is always seen as a pointer typed with an IID. The pointer itself poi
pointer that points to a table that holds the addresses of the implementation of each member f
interface. This binary structure, illustrated in Figure 3, is a core standard of COM, and all of CC
depend upon this standard for interoperability between software components written in arbitra
long as a compiler can reduce language structures down to this binary standard, it doesn't mat
programs a component or a client--the point of contact is a run-time binary standard.

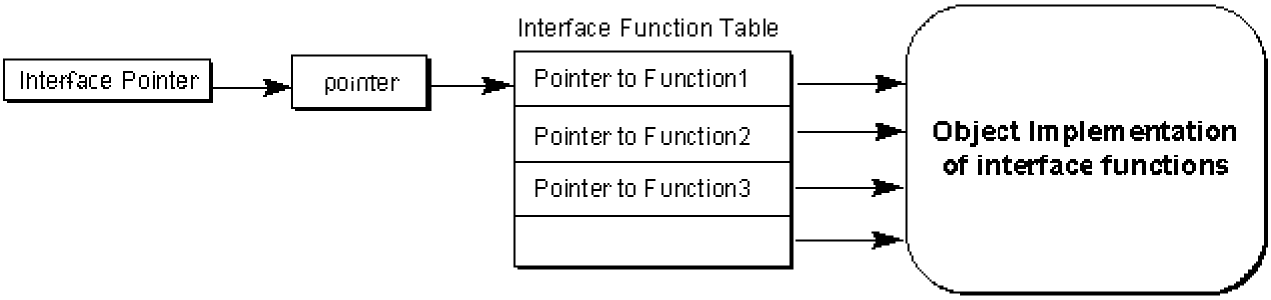


Figure 3. The binary interface structure

It is this exact interface structure that provides the ability to marshall one of these pointers be
and machines, as described in the section titled "Local/Remote Transparency."

To "implement an interface" on some object means to build this exact binary structure in mem
the pointer to the structure. This is what we want instead of having clients do it themselves! Y
assembly language if you want, but higher-level languages, especially C++, build the structure
In fact, the interface structure is, by design, identical to that used for C++ virtual functions--p
COM and OLE in C++ is highly convenient.

This is also why COM calls the table portion the "vtable" and the pointer to that table "lpVtbl."
interface is a pointer to **lpVtbl**, which points to the vtable. Because this is what C++ expects t
interface member given an interface pointer is just like calling a C++ object's member functior
have a pointer to **ISpellChecker** in the variable *pSC*, I can call a member like this:

```
pSC->LookUpWord(pszWord);
```

Because the interface definition describes all the argument types for each interface member fu
compiler will do all the type checking for you. As a client, you never have to define function pr
these things yourself.

In C, the same call would look like this, with an explicit indirection through **lpVtbl** and passing
pointer as the first argument:

```
pSC->lpVtbl->LookUpWord(pSC, pszWord);
```

This is exactly what C++ does behind the scenes, where the first argument is the *this* pointer.
COM and OLE in C++ just saves you a lot of extra typing.

discovered at run time using the value of the interface pointer itself. Therefore the compiler ge that calls whatever address is in the right offset in the vtable pointed to by the interface pointe value known only at run time. This means that when the kernel loads a client application, ther records" to patch up with absolute addresses. Instead, those addresses are computed at run ti using interfaces is a true form of dynamic linking to a component's functionality, but one that l complexity of using **GetProcAddress** and all the risks of using import libraries.

Finally, note that interfaces are considered "contracts" in the sense that when an object impler interface it must provide at least default or do-nothing code *for every member function in the* vtable element must contain a valid function pointer. Therefore a client who obtains an interfa call any member function of the interface. Granted, some member functions may just return a implemented" error code but the call will always occur. Therefore clients need not check for NU must they ever provide their own default implementations.

Implementation Location and the Registry

When a client wishes to use an object, it always starts by asking COM to locate the object's cla request the server to create an object, and return an initial interface pointer back to the client. the client can obtain additional interface pointers from the same object through the **IUnknown::QueryInterface** member function.

For some of its own native classes OLE provides specific creation APIs to streamline the initiali But for all custom components (those that are not implemented in OLE itself), COM provides a API, **CoCreateInstance**, that instantiates an object when given its CLSID. We saw earlier how this function:

```
if (SUCCEEDED(CoCreateInstance(clsID, NULL, CLSCTX_SERVER, IID_ISpellChecker
    , (void **)&pSC)))
```

The client passes a CLSID, some flags, and the IID of the initial interface. On output the pointe passed by reference in the last argument receives the pointer to the interface.

Internally, COM takes care of mapping the CLSID to the server, loading that server into memo the server to create the object and return an interface pointer. If the object is in a different pr client, COM automatically marshalls that pointer to the client's process. The basic process of ob instantiation is shown in Figure 4.

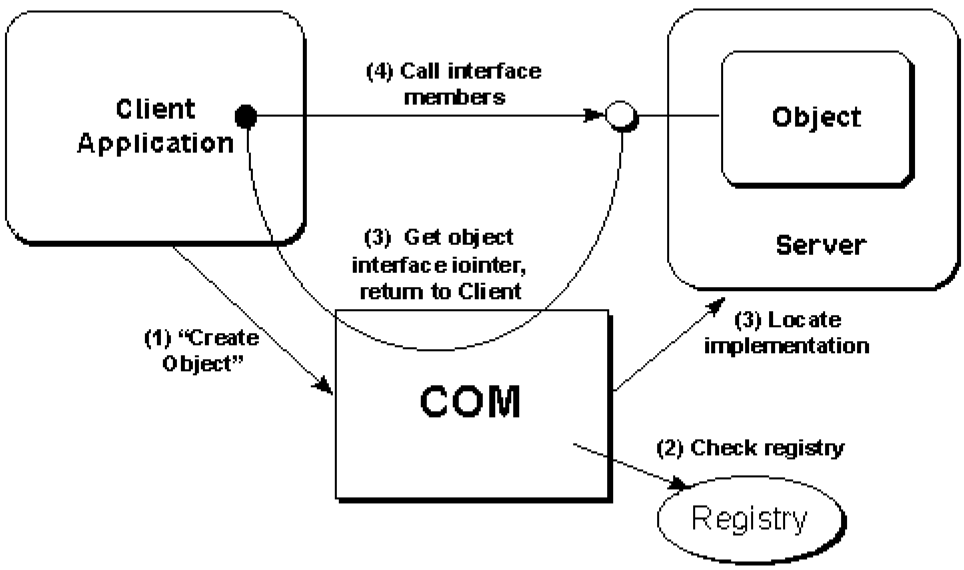


Figure 4. Locating and activating an object

COM looks in the registry to map the CLSID to its server. Servers implemented as Win32 DLLs servers") are registered as follows, assuming {01234567-1234-1234-1234-012345678AB} is t

```
HKEY_CLASSES_ROOT
    CLSID
        {01234567-1234-1234-1234-012345678AB}
            InprocServer32 = <path to server DLL>
```

To load the DLL into memory, COM needs only to call **LoadLibrary**. A "local server," on the oth implemented when an .EXE uses the **LocalServer32** key instead of **InprocServer32**. For the the **CreateProcess** Win32 API to launch the .EXE, which then initializes COM in its own proces this happens, COM in the server's process connects to COM in the client's process.

Entries for a "remote server" (one that runs on another machine) include the machine name as COM activates such a server it communicates with a resident "service control manager" (SCM) on the other machine. The SCM then loads or launches the server on the server machine and s marshalled interface pointer back to the client machine and process.

The net result in all three cases is that the client has an interface pointer through which it can object's services. We're now ready to see how COM's Local/Remote Transparency makes the pr model identical for all three cases.

Local/Remote Transparency: Marshalling and Remoting

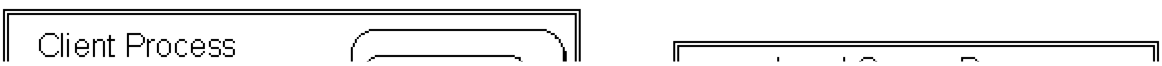
When you think of pointers to objects--such as objects implemented in C++--you normally dor about passing such a pointer to another process. Ridiculous, right? Well, this is exactly what C Local/Remote Transparency allows you to do, if the pointer is an *interface* pointer (which, conv generated from a C++ object pointer quite easily, if the C++ class is itself derived from an inte

When an in-process object is involved, COM can simply pass the pointer directly from the objec because that pointer is valid in the client's address space. Calls through that pointer end up dir object code, as they should, making the in-process case a fast calling model--just as fast as us

Now, COM cannot, obviously, just pass the object's exact pointer *value* to other processes whe remote objects are involved. Instead, the mechanism called "marshalling" builds the necessary communication structures. To "marshall" a pointer means *to create a "marshalling packet" con necessary information to connect to the object's process.* This packet is created through the CO **CoMarshalInterface** function. The packet can then be transported (through any means availa client process, where another function, **CoUnmarshalInterface**, turns the packet into an inte the client's process), which the client can then use to make calls.

This "marshalling" sequence creates a "proxy" object and a "stub" object that handle the cross communication details for that interface. COM creates the "stub" in the object's process and ha manage the real interface pointer. COM then creates the "proxy" in the client's process, and co stub. The proxy then supplies the interface pointer that is given to the client. Those familiar wi recognize this proxy/stub setup as being the same architecture used in raw RPC.

Of course, this proxy does not contain the actual *implementation* of the interface. Instead, eac function packages the arguments it receives into a "remoting" packet and passes that packet t remote procedure calls. The stub unpacks these arguments, pushes them on the stack, and ca object (using the interface pointer the stub is managing). The object executes the function anc output. The output is packaged up and sent back to the proxy, which unpacks the output and r client. This process is illustrated in Figure 5, which shows the differences between in-process, l remote cases. The figure also shows that the client only sees in-process objects, which is why "transparency"--remoting a call is transparent to the client.



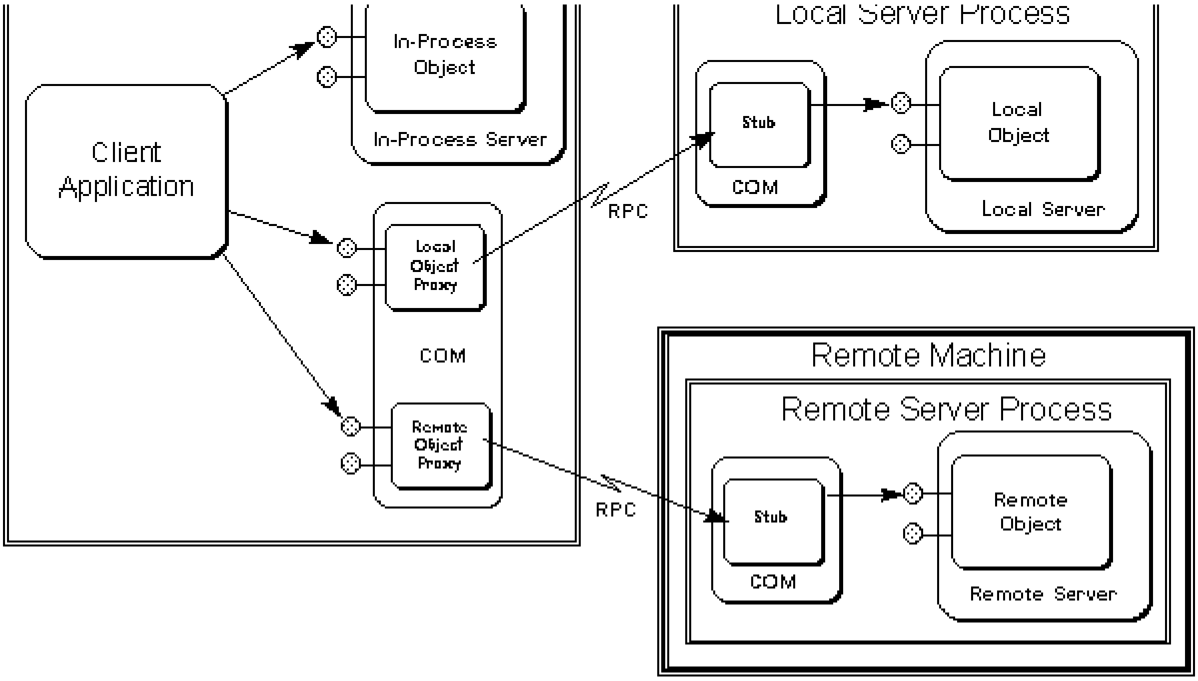


Figure 5. Local/Remote Transparency

An important feature of this architecture is that if the object's process is terminated abnormall for that object are destroyed. The proxies that exist in client processes remain active, but are "disconnected." Therefore a client can still call the proxy as always and not risk crashing--the returns a "disconnected" error code. In addition, if a client process is terminated abnormally, t disappears and its associated stub detects the disconnection. The stub can then clean up any to the object on behalf of the client that went away.

The necessary proxy and stub code for most "standard" interfaces (those defined by Microsoft) system. If you define your own "custom" interface, you must supply your own proxy/stub code the MIDL compiler described earlier will generate this code for you from an IDL file. You need c code into a DLL and you have remoting support for that custom interface.

It is important to note in the remote case that *it is not necessary to have an OLE implementat machine in order to interoperate with COM on the client machine.* All remote interface calls are with DCE-compatible RPC such that any DCE-aware system can receive the calls and convert t object architecture, be it COM, CORBA*, or any other.

*Common Object Request Broker Architecture

IUnknown: Reference Counting and Multiple Interfaces

As discussed earlier, each interface can be thought of as representing a "feature" or a "design given object will be a combination of one or more features or patterns, where the combined fu those features is what defines the object's own "component category," described with a Categ GUID). (Until recently, there were so few categories defined that CATIDs were not used to ide rather, specific registry keys like **Insertable** and **Control** were used. Microsoft has now publis specification for CATID-based categorization.)

This *ability of an object to support multiple interfaces* is a key COM/OLE innovation. No other o this concept at its core. Yet it is precisely the idea of *multiple interfaces* that *solves the versio*

The capacity for multiple interfaces depends on the **IUnknown** interface, which is the core int

COM and OLE. In fact, all other interfaces are derived from **IUnknown** and therefore are poly
IUnknown. That is, the three **IUnknown** member functions are universally the first three me
interface:

- **AddRef** increments the object's reference count.
- **Release** decrements the object's reference count, freeing the object if the count become
- **QueryInterface** asks the object to return a pointer to another interface given its IID.

The **AddRef** and **Release** members together provide for lifetime management of an object. Ev external reference to any of the object's interfaces carries a reference count that allows multip independently use the same instance of an object. Only when all clients have released their re will the object destroy itself and free its resources. As described in the last section, remoting s automatically clean up the reference count for any client process that terminates abnormally w releasing its references.

QueryInterface is what makes multiple interfaces possible. Once a client obtains the *initial* in to any object, it obtains another interface pointer to the *same* object through **QueryInterface** pass the IID of the interface you want. In return, you get back the interface pointer, if it is ava error code that says, "That interface is not supported." If you get a pointer, you can call the m of that interface, calling **Release** when you're through with it. If not, you can't possibly call me interface! Thus the object easily protects itself from unexpected calls.

QueryInterface, which is always available through every interface pointer, is how a client ask you support the *feature* identified by this IID?" By asking, a client determines the *greatest con* interfaces that both it and the object understand. This completely avoids the "least common de problem described earlier, and also completely solves the versioning problem.

The Solution to Versioning

So how does the idea of multiple interfaces solve the real-world versioning issues? Consider th

- How does one quantify the difference between any two versions?
- How does a client request a specific version?

Multiple interfaces provide precise answers to these questions. Any given revision of an object certain set of interfaces--between versions, interfaces may be added or removed, or new versi interfaces may be introduced (the old interfaces cannot be *changed*, but an object would suppo new versions of the interface). **QueryInterface** allows a client to check for whatever "version" interfaces it requires.

A critical feature of the COM programming model is that a client *must* call **QueryInterface** to interface pointer from the object (creation calls like **CoCreateInstance** by definition have buil **QueryInterface** calls). In other words, COM forces clients to always check for the presence of before ever trying to invoke members of that interface, resulting in this sort of code:

```
//Assume pObj is an IUnknown * for the object.
if (SUCCEEDED(pObj->QueryInterface(IID_<xxx>, *pInterface)))
    {
    //Call members of IID through pInterface, then:
    pInterface->Release();
    }
else
    {
    //Interface is not available, handle degenerate case.
    }
```

A client might also be written to query an object for certain interfaces when the object is first c pointers to avoid having to call **QueryInterface**

In contrast, the DLL programming model is either far more complex, costly, or risky. If you prc
import libraries, you have no way of checking for functional support in an object before invokin
so you risk "Call to undefined dynlink" errors. If you program with **GetProcAddress**, you can a
exactly what interfaces provide by default--grouping functions together as "feature" and maint
say whether or not those features are available.

> In short, COM universally simplifies the most sophisticated and difficult programming mo
> at great cost and for a single application, one could possibly create with **GetProcAddres**

A concrete example will illustrate exactly how the **QueryInterface** idea solves the versioning
Consider a COM-based specification for the category SpellCheck1.0, which says a spell-checker
BasicSpell1.0 implements the **ISpellChecker** interface. We also have AcmeNote1.0, the client
BasicSpell1.0 through this interface, as shown in Figure 6. At this point, there is little tangible
between the COM design and the original DLL design with an exported function.

Figure 6. A version 1.0 object and a client that uses it

But we want to write the SpellCheck2.0 specification to add the custom dictionary feature that
AddWord and **RemoveWord** functions. (A full design would include a way to enumerate all th
dictionary, but a discussion of such a design is beyond the scope of this paper.) With a COM de
introduce these functions in a new interface, but we have great flexibility in how we do this:

- Introduce **ISpellChecker2**, which is derived from **ISpellChecker** but has an entirely dis required. Objects would implement this new interface, which would then implement **ISp** advantage here is that the two interfaces end up sharing the same vtable, thereby reduc memory overhead. This is important when the category being defined typically leads to h thousands of object instances at run time. By sharing the vtable, you save 16 bytes per i **lpVtbl** pointer plus the **IUnknown** entries in the table. Custom controls are an example savings are important. A spell-checker, on the other hand, doesn't need to worry about t will usually have only a single instance per process. So this design isn't advantageous ov
- Introduce **ICustomDictionary**, which contains the two new functions (plus the **IUnkno** Objects would implement this as another interface alongside **ISpellChecker**. This is the design choice.
- Introduce two new interfaces, such as **ICustomDictionaryAdditions** and **ICustomDict** to separate the features of adding words and editing the contents of the dictionary. In th case there is little to gain from this separation; however, in other designs different functi different interfaces. One usually wants to avoid going to the extreme of defining several each have a single member function. This complicates client programming by forcing two per function: one to **QueryInterface**, one to the member, one to **Release**, depending o are cached.

For the purposes of our discussion, we'll choose the second option and define **ICustomDiction**
IID:

```
[uuid(8e47bfb0-633b-11cf-a234-00aa003d7352), object]
interface ICustomDictionary : IUnknown
    {
    HRESULT AddWord(OLESTR *pszWord);
    HRESULT RemoveWord(OLESTR *pszWord);
```

}

Now that we have the SpellCheck2.0 specification, the developers at Basic Software and Acme some point, each decide to upgrade their respective products to fit this new specification. Earli is very difficult with the plain DLL model to have both companies upgrade their products indep this is exactly what COM was designed to support! COM allows either company to add support interface without risking compatibility with the old version of the other.

Let's say that Basic Software chooses to release BasicSpell2.0 first, and the spell-checker obje both **ISpellChecker** and **ICustomDictionary**. In making this change, the developers don't ha anything in the existing **ISpellChecker** code except its **QueryInterface** implementation. The code simply continues to exist as it always has, and COM's multiple interface design encourage

Now when AcmeNote1.0 encounters this new object, it sees exactly what it did with BasicSpell Figure 7. Because AcmeNote1.0 doesn't know about **ICustomDictionary**, BasicSpell2.0 appea the same as BasicSpell1.0. That is, the two are perfectly polymorphic, and no compatibility pro

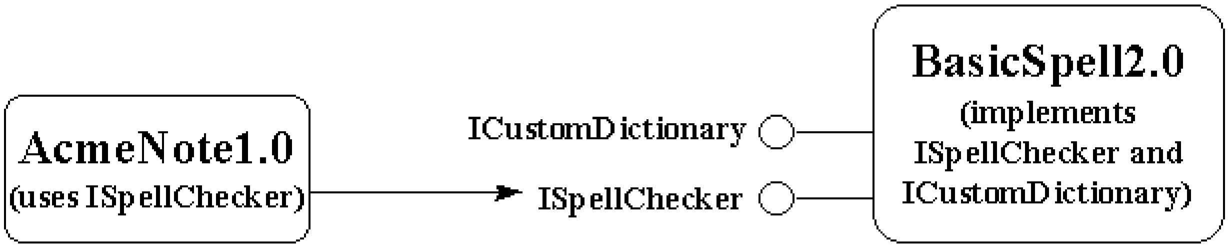


Figure 7. An old client sees a new object exactly as before, because new features sho interfaces.

Now let's suppose the reverse happens, that Acme Software releases AcmeNote2.0 first. Acme written to enable a custom dictionary feature if the spell checker component it finds supports t this point, only BasicSpell1.0 exists. But because AcmeNote2.0 must call **QueryInterface** for **ICustomDictionary** in order to use that feature, *it must be coded to expect the absence of th* is, if a client requests the **ICustomDictionary** interface when initially creating the object, tha and no object is created. A client can specify this interface on creation if it wishes to work *only* objects. The client would then provide a meaningful error message, far more meaningful than undefined Dynlink." The COM programming model forces clients to handle both the presence *a* of a particular interface. Doing this is not difficult, but because you *must* call **QueryInterface** prepared. So when AcmeNote2.0 encounters BasicSpell1.0, as illustrated in Figure 8, its reque **ICustomDictionary** fails and AcmeNote simply disables its custom dictionary features.

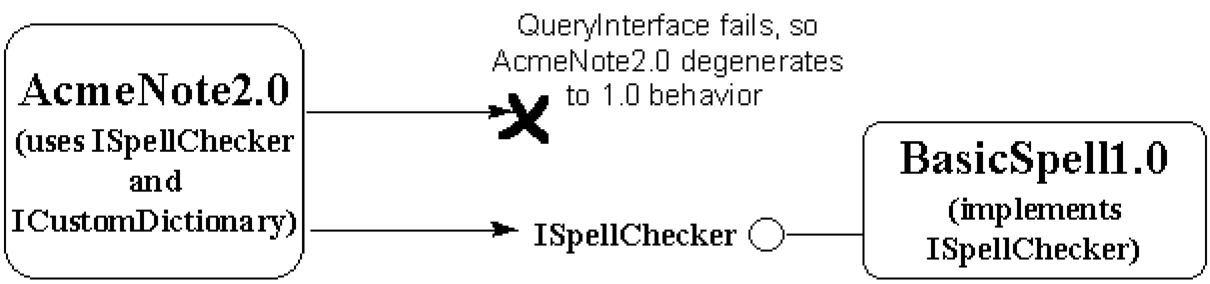


Figure 8. A newer client encounters an older object and degenerates gracefully.

As we saw earlier, the easiest DLL programming model encourages clients to expect the *prese* functions. Catastrophic failure results when those functions are absent, but because it takes e handle such a case robustly, few client programs actually bother. In marked contrast, the COM model forces a client to expect the *absence* of interfaces. As a result, clients are written to rob

What makes the COM model even more powerful is that when a client discovers the *presence* o achieves richer integration with the object. That is, expecting the absence of interfaces means works in a degenerate situation (as opposed to failing completely). When interfaces exist on ar just get better.

To demonstrate this, let's say that AcmeNote2.0 is written so that it creates an instance of the object *only* when the user invokes the Tools/Spelling command. So when users do a spell-chec BasicSpell1.0 installed, they'll get the basic features. Now let's say that *without closing AcmeN installs BasicSpell2.0*. The next time the user invokes Tools/Spelling, AcmeNote2.0 finds that **ICustomDictionary** is present and so it enables additional features that were not available be once AcmeNote2.0 discovers the availability of the feature, it can enable richer integration bet component, as shown in Figure 9. Older clients, like AcmeNote1.0, still continue to work as the

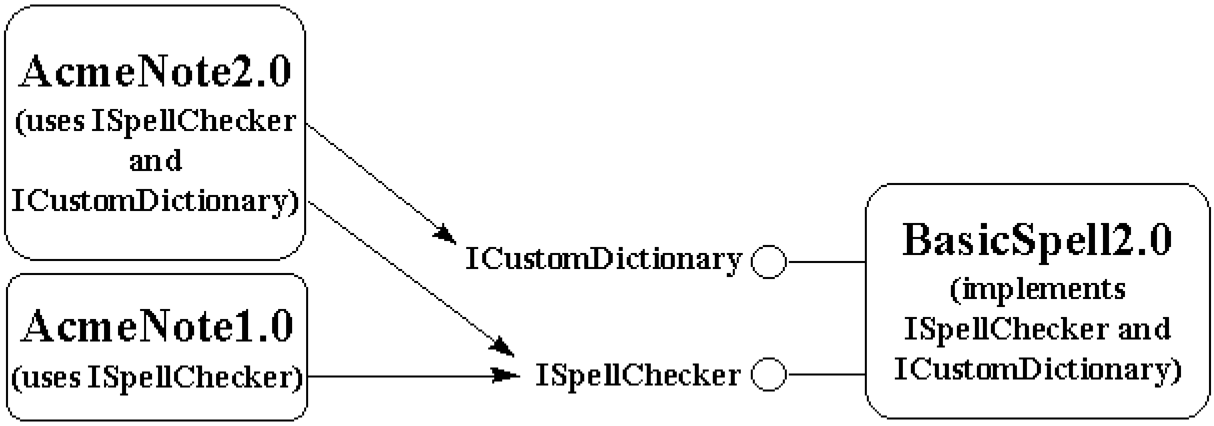


Figure 9. When an upgraded client meets an upgraded component, they integrate on a than before.

This improvement in integration and functionality is *instant* and happens in a *running* system. mean when we say that COM is meant to solve component software problems in a *binary runn* without having to turn everything off, we can install a new component and instantly have that integrate with already running clients, providing new features immediately.

Earlier in this article we saw what happened when an older version of a DLL accidentally overw version: Any clients that were implicitly linked to the newer version started to show "Call to un and wouldn't run at all. We can see here how COM completely solves this problem. Suppose a another application that itself installs BasicSpell1.0 and accidentally overwrites BasicSpell2.0. raw DLLs and AcmeNote2.0 had implicit links to BasicSpell2.0, AcmeNote2.0 would no longer r using COM components, AcmeNote2.0 will run just fine, and when the user invokes Tools/Spell AcmeNote2.0 will simply not find **ICustomDictionary** and thus will not enable its custom dicti Again, the client expects the *absence* of interfaces and is prepared to work well without them. because interfaces completely avoid implicit linking, the kernel will never fail to load the applic

It should be obvious now that COM's support for multiple interfaces solves the versioning probl the first revision. But the same solutions also apply over many subsequent versions, as illustra 10. Here we have BasicSpell4.0, which contains **ISpellChecker**, **ICustomDictionary**, **ISpecialtyDictionaries**, and maybe even **IThesaurus** and **IGrammar** (extra features, no les

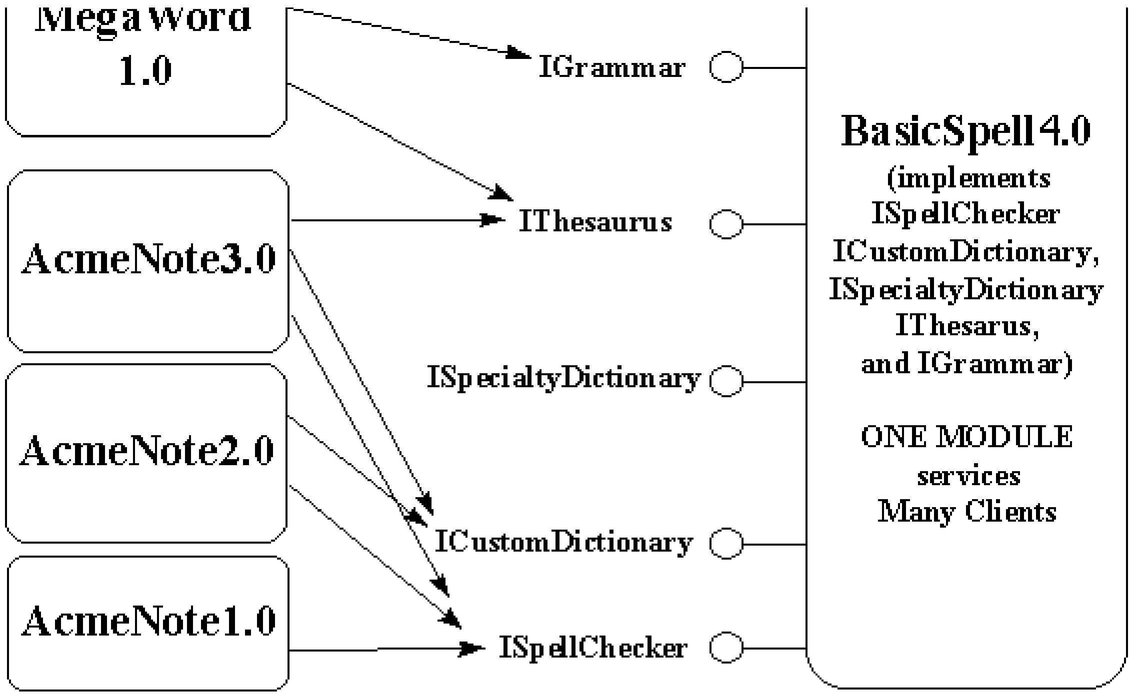


Figure 10. One object can serve many different clients expecting different versions.

This simple version of the object now supports clients of the SpellCheck1.0 specification, client SpellCheck2.0 specification, and those written to specifications that involve the other three add Some of those clients may only be using the thesaurus features of this object and never bothe like **ISpellChecker**. Yet the single object implementation supports them all.

In addition, this new object server can overwrite any old version of the server. Earlier in this p that building components in the DLL model typically leads to redundant code stored in multiple different DLL versions, like VBRUN100.DLL, VBRUN200.DLL, and so on. COM allows you to fact differences from version to version in such a way that you can keep the old code available to o provide new code to new clients all within one server module. Hard drive manufacturers may b users will be happy!

Of course, the newer modules will be larger than the old ones, simply because there's more co added interface. However, because all the code is encapsulated behind interfaces, the object ca different implementation tricks to (1) reduce the basic module size, (2) improve performance, minimize memory footprint. In COM, an object is not required to allocate an interface's functio interface is requested--each version of an object can thus choose to initially instantiate only th that are used frequently, leaving others out of memory completely until needed. Code-organiza help keep the interface code itself out of memory until it is needed. Such optimizations can mi server's load time because only a few code pages are marked "preload." Alternatively, the deve choose to place the rarely used code in another module altogether, using the object-reuse tech *aggregation* to instantiate interfaces from that module only when necessary, while still making appear as part of the object itself.

Over time, the code for the less-frequently-used interfaces will migrate from being placed in lo pages in the object module to being separated into another module, to being discarded altoget interfaces become obsolete. At such a time, very old clients may no longer run well, but becau written to expect the absence of interfaces, they'll always degenerate gracefully.

The bottom line is that COM provides a complete solution for versioning problems, at the same a clear and robust migration path for code to gradually move farther out of a working set as it frequently used. Even when the code is discarded completely, the COM programming model gu degeneration. COM offers a truly remarkable means for removing obsolete and dead code from

COM vs. "Objects"

There is often a fair bit of debate in the popular trade media about whether or not COM and OL or compete with "objects" or "object-oriented programming." Some say that OLE doesn't suppc objects, whatever that means. Some think OLE competes with object-oriented programming (C like C++, or with frameworks like MFC. All of these claims are untrue, because COM and OLE w complement existing object technologies, as described in the next two sections.

COM/OLE's Problem Space

COM and OLE do not compete with object-oriented languages nor with frameworks because the problems in a domain that is not only separate from the problem space for languages and fram complementary, as illustrated in Figure 11.

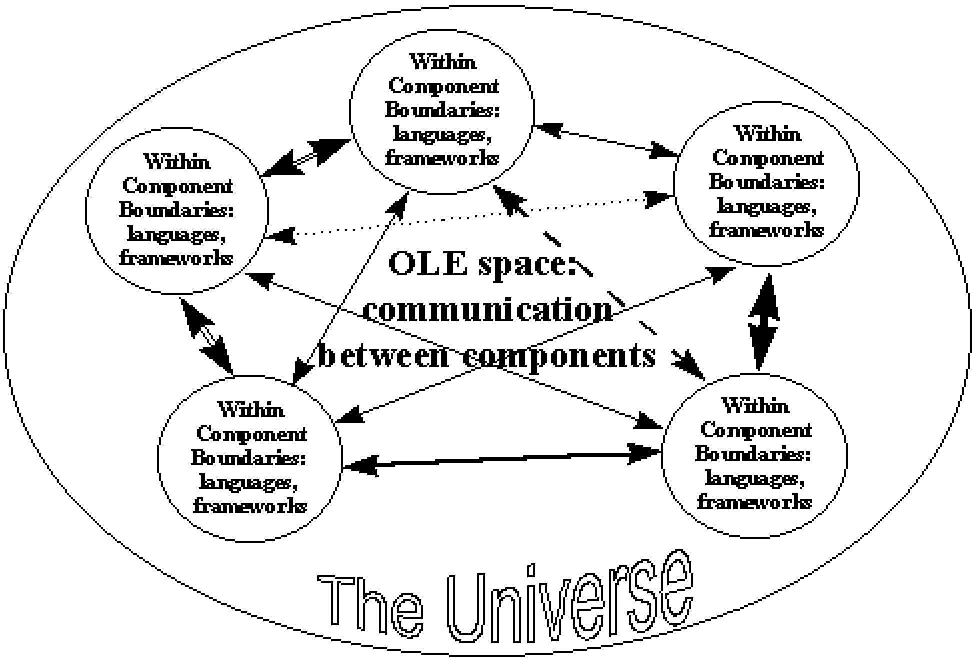


Figure 11. COM and OLE's problem space

Languages and frameworks are absolutely fabulous tools for programming the *internals* of com objects. However, when you want different objects written by different people to interoperate, across time (of deployment) and space (objects running in other processes and running on oth COM and OLE provide the means for doing so.

In fact, it is easy to see how one might take a language (such as C++) -based object and crea wrapper code that factors the language object's interface into COM-style interfaces, as shown i doing so, that language object is suddenly available to use from any other client process throu mechanisms. "From CPP to COM" by Markus Horstmann (MSDN Library, Technical Articles, Win OLE Articles, Component Object Model) is a good paper on creating this wrapping layer for C+ turning the C++ "interface" into COM interfaces.

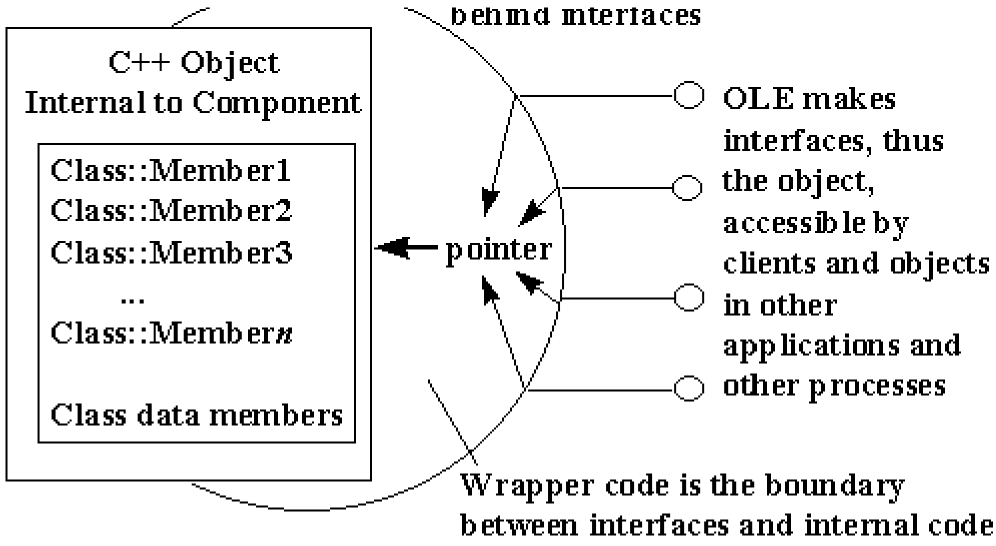


Figure 12. Wrap a language-based object inside interfaces to make the object availabl processes.

To provide OLE support, the framework classes of MFC turn the more raw and complex OLE int certain high-level features (such as compound documents and OLE Controls) into a simplified (easier to work with in application programming. In such cases the interface designs existed firs developers created a C++ class for them.

So although object-oriented languages are great ways to express object-oriented concepts in s great ways to express *implementation details in source code*, OLE is concerned with the *comm* between pieces of *binary code in a running system*. Frameworks like MFC make the boundary b worlds simpler by providing default implementation as well as wrapping the communication lay COM and OLE are to frameworks like MFC as mathematics is to a calculator: A calculator make: operations effortless but you still have to understand *why* you are using those operations; in th MFC makes certain OLE features very simple to incorporate into an application, but you still ha understand *why* you want those features!

Object Concepts in COM / OLE

If you can take a language-based "object" and wrap it in COM/OLE interfaces, is it still an "obje from a COM/OLE client? Does a client see an entity that is *encapsulated*? Does that entity supp *polymorphism?* Does that entity support *reuse*? These are the fundamental concepts behind th "object-oriented programming," and COM, at its core, supports all three. Objects in COM/OLE b objects in all the ways that matter. The *expression* of the particular concepts is a little differen ends are achieved.

Encapsulation

How does COM support *encapsulation*? Easily. All implementation details are hidden behind the structures, exactly as they are in C++ and other languages: The client sees only interfaces an about object internals. In fact, COM enforces a stricter encapsulation than many languages bec interfaces *cannot* expose public data members--all data access must happen through function language extensions for compilers can let you express public members in *source code*, but on t all data is exchanged through function calls, which is widely recognized as the proper way to d

Polymorphism

How does COM support *polymorphism*? This happens on three different levels:

- First, two interfaces that derive from the same base interface are polymorphic in that ba: as they would be in C++. This is because both interfaces have vtable entries that look e› of the base interface. The primary reason for this is that all interfaces are derived from **I** thus look exactly alike in the first three entries of the vtable. When you program COM an you actually use C++ inheritance to express this polymorphic relationship. When prograr interface structures are defined as explicit structures of function pointers where polymor| structures share the same set of initial entries in the structure.
- Second, object classes (and instances) that support the same interface are polymorphic i That is, if I have two object classes that both support an interface like **IDropTarget**, the client code that manipulates the **IDropTarget** of one object class can be used to manipu **IDropTarget** implemented in another class. Such client code exists in the OLE system-l drop service, which uses this interface to communicate with any potential recipient (an a control, or even the desktop itself on Windows 95) of a drop operation.
- Third, object classes that support the same *set of multiple interfaces* are polymorphic ac *set*. Many service categories expect this. The OLE specification for embeddable compoun objects, for example, says that such objects always support the **IOleObject**, **IViewObje** **IDataObject**, and **IPersistStorage** interfaces, as well as a few others. A client written specification can host any embeddable object regardless of the type of content, be it a ch sound bite, table, graphic, text, or other. All OLE Controls are also polymorphic in this m supporting a wide range of capabilities behind the same set of polymorphic interfaces.

Reuse

How does COM/OLE support *reuse*? The reader might ask, "Don't you mean *inheritance*?" No--I must be understood that *inheritance* is **not** a fundamental OOP concept; inheritance is how on polymorphism in a programming language and how one *achieves* code reuse between classes. *means to polymorphism and reuse*--inheritance is not an *end* in itself. Many defend inheritance of OOP, but it is nothing of the sort. Polymorphism and reuse are the things you're really after.

We have already seen how COM supports polymorphism. It supports reuse of one component b through two mechanisms. The first, *containment*, simply means that one object class uses ano internally for its own implementation. That is, when the "outer" object is instantiated, it intern an object of the reused "inner" class, just as any other client would do. This is a straightforwar relationship, where the inner object doesn't know that its services are being used in the impler another class--it just sees some client calling its functions.

The second mechanism is *aggregation*, a more rarely used technique through which the outer interface pointers from the inner object and exposes those pointers through its own **QueryInt** saves the outer object from having any of its own code support an interface and is strictly a *co* make certain kinds of containment more efficient . This does require some extra coding in the make its **IUnknown** members behave as the outer object's **IUnknown** members, but this am few lines of code. This mechanism even works when multiple levels of inner objects are in use, outer object can easily obtain an interface pointer from another object nested down dozens of l

What OLE Is Made Of

We have seen the reasons why the designs in COM exist and the problems that COM solves. W how COM and OLE complement OOP languages and frameworks, and how COM supports all the object concepts.

So what, then, is "OLE" itself? Why do people make such a fuss about OLE being "big and slow "difficult to learn"? The reason is that Microsoft has built a large number of additional services specifications on top of the basic COM architecture and COM services (like Local/Remote Trans| by itself, COM is simple, elegant, and absolutely no more complex than it has to be. With the r any developer can learn the core of COM in a matter of days and see its real potential.

However, all the layers of OLE that are built on top of COM are overwhelming. There are more
functions and perhaps 120 interface definitions averaging 6 member functions each. In other v
thousand bits of functionality that continues to grow! Thus, trying to understand what everythi
daunting effort--I spent a few years doing this myself in order to produce the books *Inside OLE*
Inside OLE, 2nd Edition (1995) for Microsoft Press. I refer the reader to those titles (primarily t
edition) for a comprehensive and incremental approach to understanding all the pieces of OLE.

For the purposes of this paper, let me provide an overview of this approach to understanding. S
see that *all functionality in OLE falls into three categories*:

1. API functions and interfaces to expose OLE's built-in services, including a fair number of and helper objects.
2. API functions and interfaces to allow customization of those built-in services.
3. API functions and interfaces that support creation of "custom services" according to varic specifications.

The following sections describe each of these in a little more detail.

OLE's Native Services

One reason OLE seems so large is that it offers quite a wide range of native services (we've alr
some of these):

- Implementation location
- Local/Remote Transparency ("remoting" and "marshalling")
- Task memory allocation
- Structured storage (described in detail below)
- File, Item, Composite, Pointer, and Anti Monikers
- The "running object table"
- Type library and type information creation and management
- Type conversion routines
- Clipboard data exchange
- Drag-and-drop data exchange
- Data caching
- Default "handling" for embedded objects
- Helper and wrapper functions, including support for string and array types.
- Helper objects such as standard dispatch, font, and picture objects, and "advise" holders

As we've already seen, the services of implementation location, marshalling, and remoting are
fundamental to the ability of objects and clients to communicate across any distance. As such,
are a core part of a COM implementation on any given system.

Other services are not quite as fundamental. Rather, they are implementations of a standard a
higher-level interoperability. For example, the "Structured Storage" standard describes a "file s
file" service to standardize how software components can cooperate in sharing an underlying d
such sharing is essential to cooperation between components, OLE implements the standard in
"Compound Files." This eliminates all sorts of interoperability problems that would arise if ever
client application had to implement the standard itself.

To see why a standard like this is important, let us look briefly at how today's file systems the
about. When computers first gained mass-storage devices, there weren't these things we call "
systems." What ran on the computer was *the* application, which did everything and completely
system resources, as shown in Figure 13. In such a situation, the application simply wrote its c
wanted on the storage device, itself managing the allocation of the sectors. The application pre
the storage device as one large contiguous byte array in which it could store information howe

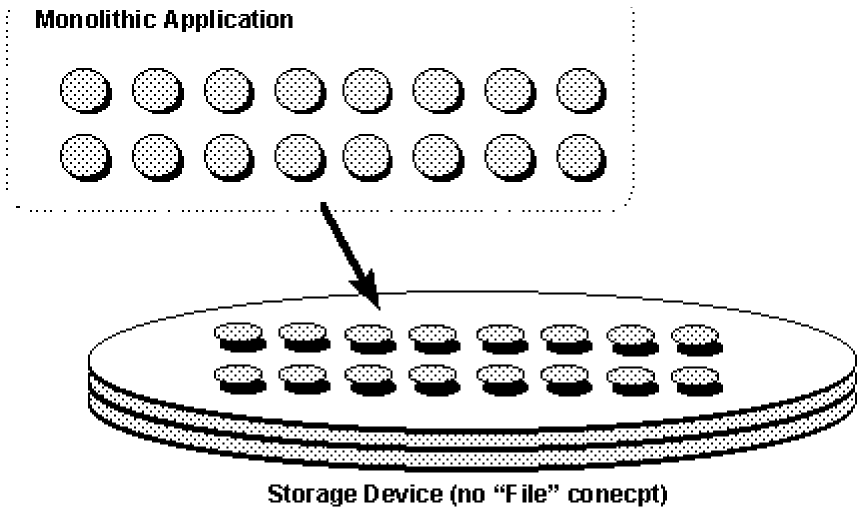


Figure 13. When there was only one "application" on a machine, that application own resources, such as the mass-storage device.

Soon it became desirable to run more than one application on the same machine. This requirec agent to *coordinate the sharing* of the machine's resources, such as space on the mass-storage Hence, file systems were born. A file system provides an abstraction layer above the specific a sectors on the device. The abstraction is called the "file" and is made up of usually non-contigu the physical device, as shown in Figure 14. The file system maintains a table of the sequence c make up the "file." When the application wishes to write to the disk, it asks the file system to c which is to say, "Create a new allocation table and assign it this name." The application then s a continuous byte array and treats it as such, although physically the bytes are not contiguous way the file system controls cooperative resource allocation while still allowing applications to as a contiguous byte array.

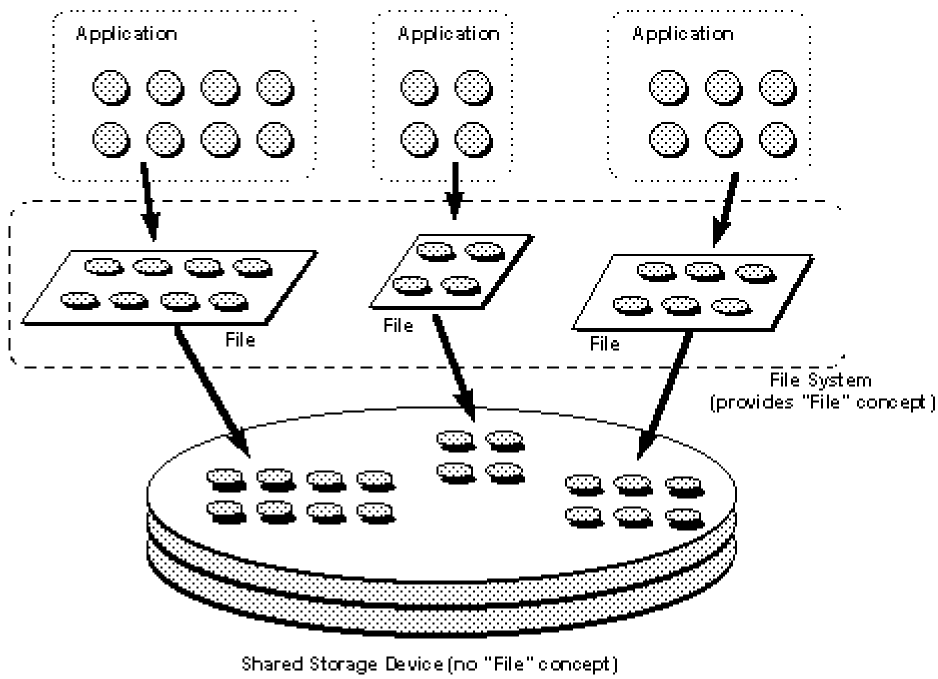


Figure 14. A file system allows multiple applications to share a mass-storage device.

File systems like this work great as long as all parts of the application *itself* are coordinated. In software paradigm, this is no longer true! Applications might be built from hundreds of compor developed at different points in time and space. How, then, can these components work coope

through some standardized interfaces so they can negotiate about who gets what part of the fi
obvious that such a design would be extremely fragile and horrendously slow.

A better solution is to repeat the file system solution--after all, the problem is the same: differ
code must cooperate in using an underlying resource. In this case the underlying resource is a
storage device, although conceptually the two are identical. The OLE Structured Storage specif
the abstraction layer that turns a single file into a file system itself, in which one can create di
elements called "storage objects" (that implement **IStorage**) and file-like elements called "str
(that implement **IStream**), as shown in Figure 15.

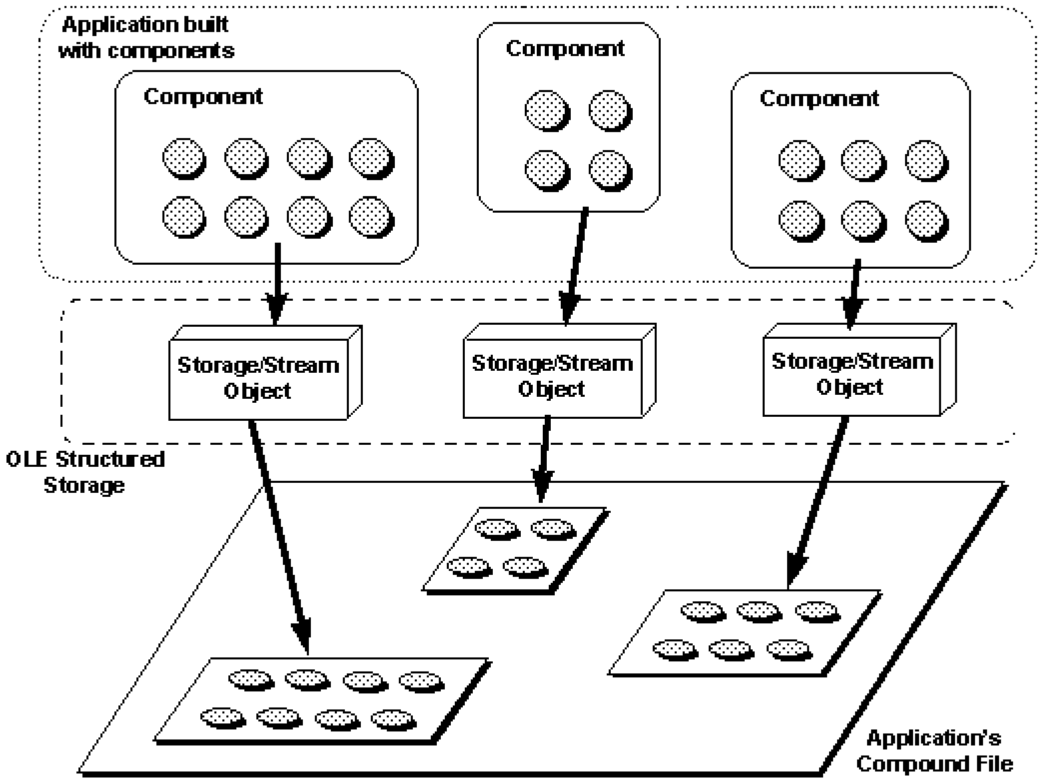


Figure 15. OLE's Structured Storage allows multiple components to share the same fil

OLE's implementation of this standard controls the actual layout of data bits inside the file. The
components themselves, however, see stream objects as contiguous byte arrays once more, sc
you have working with files translates directly to working with streams. In fact, there is a one-
correspondence between typical file-system APIs such as **Read**, **Write**, and **Seek**, and the me
of the **IStream** interface through which one uses a stream object.

Structured Storage eliminates the need for each component in the application to negotiate its s
requirements; instead, this is coordinated in OLE's implementation. The only necessary bit tha
must communicate to components is to pass each of them an **IStorage** or **IStream** pointer, d
object's persistence mechanism. If an object implements, for instance, **IPersistStorage**, it is
would like to have an **IStorage** as a basis for its storage needs. Within the storage object it th
storages and streams as necessary. If an object implements **IPersistStream**, on the other ha
it only needs a single stream for its storage needs.

Customization of Native Services

Some of the functions in the OLE API and some of the interfaces deal with customizations to O
services. One example will suffice to describe this "customization."

OLE's implementation of the Structured Storage model described in the last section typically w underlying file on a file system. However, you can customize this behavior by redirecting the I/O to another storage device through what is called a "LockBytes" object (one that implements **ILoc** object knows how to do is read or write a certain number of bytes in a certain location on some By default, OLE itself uses a LockBytes that talks to a file on the file system. However, you are implement your own LockBytes (or use OLE's other implementation that works on a piece of gl redirect data to a database record, a section of another file format, a serial port, or whatever. allow you to install your LockBytes underneath the storage implementation. When someone wr stream through **IStream::Write**, for instance, that data ultimately shows up in your **ILockBy** code, allowing you to place it anywhere you want.

Custom Services

OLE itself can only provide a limited number of native services--specifically those that need to and standardized. A great deal of OLE's functionality is to support the creation of what I call "c that fit into one or more "service type" specifications.

There are three primary (and sometimes complex) type specifications that Microsoft has define each of which involves a large number of OLE API functions and interfaces. They are:

- OLE Documents, for the creation and management of compound documents. The specific either how you create "embeddable" or "linkable" objects or how you create a "container" objects, with in-place activation as an optional feature.
- OLE Controls, for the creation and management of custom controls. The specification des create or host any kind of custom control in an OLE model. Currently there are hundreds thousands, of vendors working on controls or control containers.
- OLE Automation, for programmability and scripting. The specification describes how you automation object or an automation "controller" that can drive that object according to a kind. Visual Basic exploits OLE Automation to a large extent.

Each "type" specification describes the general interaction between an object of the type and a an object. From the specification one can implement a wide assortment of specific objects that categories. For example, a sound bite, a video clip, a chart, a picture, a presentation, text, and examples of specific instances of the "embeddable compound document object" type. Buttons, labels, check boxes, tabbed dialogs, list boxes, scrollbars, toolbars, sliders, dials, and status lin examples of custom controls, each of which can be implemented as an OLE control. OLE Autom typically used to create objects that expose the functionality of an entire application, like Micro that a scripting engine like Visual Basic can drive that application to perform certain operations need for a user to execute the steps (that is, "automation" in its most generic sense).

In addition to these Microsoft-defined categories, other industry groups have defined service ca their particular needs. Some examples include the industries of real-time market data, health point-of-sale, and process control. Other groups are actively working on more specifications.

No matter who defines the service, the idea is that many people can implement clients or obje the specification, which describes the abstraction that either side (client or object) uses to viev that all clients are polymorphic to the objects and all objects of the category are polymorphic t

What is most important here is that *COM is an open architecture in which anyone can create a service categories without any input or involvement on Microsoft's part and without any depen Microsoft enhancing its operating systems*. This independence was intentionally built into the d COM, Microsoft has created an extensible service architecture in which anyone--*without any ce whatsoever*--can define new service categories, publish them, implement components and clie them, and deliver those components to customers.

The true "openness" of an architecture lies in its *design and extensibility,* not just in the *proce* the design occurs (which at Microsoft is *open*, just not *committee-based*!). COM and OLE enabl and asynchronous innovation, design, development, and deployment of component software, t

OLE's Life Purpose

In this paper we have seen why COM and OLE exist, the problems they were designed to solve they provides, and the open flexibility that is inherent to their design. But you're probably aski questions: *What does OLE mean to me?* and *What does OLE mean to my project?*

If there's one word to describe what OLE is all about--technically speaking--it is ***integration***. T integration means to you and the projects you're working on--do you think about integration w How about integration with other applications? Or integration with third-party add-on compone questions you ask, OLE probably has an answer for you. For example, the Windows 95 and Wir shells have extension features based on OLE components. Integration between applications, wl OLE Documents, OLE Automation, Drag and Drop, or other private protocols, almost always inv the OLE Component Object Model provides an ideal architecture for plug-ins or add-on models solves all those problems that appear across multiple versions and revisions of interfaces.

Everything you need to exploit this integration on a single machine is already in place: COM in shipped nearly three years ago and is already installed in tens of millions of desktops. Since th Microsoft and other parties have continually added new enhancements that add even more pov capabilities, such as the introduction of Network OLE in Windows NT 4.0.

All of this leads us to an even greater question: *What is it all for?* We understand what OLE is a technically, but what is OLE *about?* In what direction is OLE moving? What does OLE mean for industry as a whole? What is OLE's "life purpose"? I contend that what OLE is *really* about is m one company such as Microsoft, yet still very personal to everyone involved. OLE's purpose is t component software a working reality. Many groups have claimed to have "superior componen yet OLE's COM is the only one that solves the right problems for the right reasons.

Why all this fuss about component software? It is a long-standing dream in the software indus quickly assemble applications from reusable components, allowing us to meet customer demar quickly. Object-oriented programming to some extent brought us closer to that goal, but it is li code problems. In the real world, problems also need to be solved in a binary running system, cannot recompile and redeploy all the software in a system every time one object implementat change. OLE and COM were designed specifically to address the issues of components in a runi in running applications. This is why COM is based on a binary standard, not a language or pseu standard. COM is the glue that makes component integration work right.

But we must realize, as an industry, that the purpose of component software is more than just easier for *developers* to assemble applications, because no matter what your methodology, dev code cannot perfectly match customer needs when they arise. That is, our development metho involve asking customers what their *problems* are, which assumes they can even articulate the know the problems, we can try to formulate solutions, and after about a *minimum* three-to-six we might be able to deliver a solution. However, the current "application backlog," as it's callec not a matter of months.

The inherent problem here is that by the time we can deliver a prepackaged solution, the custo and needs have changed--often times drastically. We might deliver a solution that solves old p new ones. So we're caught in a treadmill, always trying to catch up. We try to preach this or th methodology as "the solution," forgetting that *customers want immediate solutions to their pre* Customers care about solutions, not OOP, languages, or "object purity." And if customers care then so should developers, because customers are the *single source of income in the compute*

One way we've tried to solve this problem is by creating ever more complex and generic *tools* nerve to call "applications." These do-it-all applications, with every feature imaginable, general user to *mentally map their problem to the tool.* This is exactly why software seems so hard to this mental mapping is so difficult.

So what's special about component software? It does not just help us run faster in the treadmi *software is about enabling end users to quickly and easily construct exactly the applications th their immediate problems.*

This might be done by adding a component to an existing application to provide some new cap wasn't there before. In a sense, a user who makes a small customization to an existing one ha application. Many customer problems can be solved this way, but it requires a strong compone to do it. COM is just such an architecture.

As a complement, *users should be able to express their exact problem to the computer and ha assemble the application from available components.* If that application isn't quite right, the us the shortcomings and try again. I believe that with this positive feedback loop--one that develo accustomed to--it will be possible for users to express their problem so easily that the compute the right application within a matter of minutes. When the problem is solved, the user throws away. After all, the definition of an "application" is "a particular solution to a particular problem component world, the "application" is nothing more than a list of connections between compon

Component software is about putting more power into the hands of users instead of concentrat the hands of developers. This is precisely what helps users solve their own problems. Of course many components that are also targeted towards developers as well, for many developers will higher-level components from lower-level components. Component software serves everyone!

Yet there are two requirements to make this vision of component software really work. One wil software that allows the user to naturally express a problem, and then turn it into an applicatio this will someday involve natural speech recognition.

The other requirement is a great diversity of components. Science knows that any successful s biological, ecological, societal, or technological--requires diversity, which is the natural state of Imagine a world in which the only plant is grass, the only animals are mice, and all the human Pretty scary. A diversity of components is necessary for a successful component software syste low, medium, and high-level functions targeted to all people from the low-level hack to the mo user.

But where will these components come from? Today there exists a tremendous amount of softv functionality inside large, monolithic applications. Unfortunately, because it is part of a monolit functionality is only available to that *one* application. Through a process called *componentizatio* up a large application into smaller and smaller components, thereby making that functionality many other ways. This won't happen overnight, of course--applications will probably be broken pieces first, then those pieces broken down, and those broken down, and so on. It is one of CC design points to allow this large-scale componentization to occur *gradually*, while at the same reintegration of small components into larger ones.

For example, you can first make components available from within a large .EXE that is riddled code. Then behind the scenes, you can rewrite that code and break it down into smaller and m pieces as you see fit. The COM architecture provides a smooth path through the whole compon process--from legacy code to small and fast components. You don't have to reengineer everyth benefits.

Once a monolith has been broken down into components, you should be able to reassemble th application. But because there are now possibly hundreds of components, they can also be rea thousands of other ways, thereby creating many other useful applications *from the same code* these applications will be far more suited to a specific customer task than was the general-pur and each of them will be smaller and easier to use. Customers will be able to solve their proble

Again, this is what component software is all about! Simply by virtue of programming in a com that encourages well-factored software designs, we can create thousands of useful components integrated in millions of ways in order to solve customer problems!

So where do you, today's software developer, fit into this world view? There are many opportu Microsoft cannot do it all. Microsoft's role is to provide the "plumbing" on Windows and Macintc systems, and will, of course, be trying to produce some of the best components available. But other opportunities, not only in competing with Microsoft in providing high-value components, providing the plumbing for other operating systems, in designing new services or component t integrating components for customers, and in creating the end-user tools that integrate compc automatically. There are likely to be many other opportunities that we cannot even imagine to

Should you be scared? I think not. When users are empowered to create their own applications own problems, there will be explosive growth in this industry! The computer industry history sl innovation that empowers more people to solve their own problems has generated a surge of g happened with operating systems, with compilers, with graphical user interfaces, and I believe with component software. Who knows, users might start selling their "applications" to each oth ultimately would create a larger market with even more users.

This potential for market growth is very exciting, because in an expanding universe, *everyone* in which to grow. Today's software industry (and business in general) is in a phase of serious c with a new merger announced weekly. When everyone is crowded together in a limited space, anyone to grow is to consume one another. But in an expanding market, space itself increases companies like Microsoft cannot, logistically speaking, grow fast enough to fill in the gaps crea expansion. Thus many opportunities will arise for everyone--to grow, to create new businesses to succeed.

What OLE is really *about is your success.*

Top of Page