The Component Object Model: Technical Overview

This paper is adapted from an article appearing in *Dr. Dobbs Journal*, December 1994. © 1996, *Dr. Dobbs Journal* and Microsoft Corporation. All rights reserved. Original version available at Microsoft, this version reformatted for printing.

Introduction

The Component Object Model (COM) is a software architecture that allows applications to be built from binary software components. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by OLE. OLE services span various aspects of commonly needed system functionality, including compound documents, custom controls, interapplication scripting, data transfer, and other software interactions.

Figure 1. OLE technologies build on one another, with COM as the foundation.

These services provide distinctly different functionality to the user. However they share a fundamental requirement for a mechanism that allows binary software components, derived from any combination of pre-existing customers' components and components from different software vendors, to connect to and communicate with each other in a well-defined manner. This mechanism is supplied by COM, a software architecture that does the following:

- Defines a binary standard for component interoperability
• Is programming-language-independent
• Is provided on multiple platforms (Microsoft® Windows®, Windows 95, Windows NT™, Apple® Macintosh®, and many varieties of UNIX®)
• Provides for robust evolution of component-based applications and systems
• Is extensible by developers in a consistent manner
• Uses a single programming model for components to communicate within the same process, and also across process and network boundaries
• Allows for shared memory management between components
• Provides rich error and status reporting
• Allows dynamic loading and unloading of components

It is important to note that COM is a general architecture for component software. Although Microsoft is applying COM to address specific areas such as controls, compound documents, automation, data transfer, storage and naming, and others, any developer can take advantage of the structure and foundation that COM provides.

How does COM enable interoperability? What makes it such a useful and unifying model? To address these questions, it will be helpful to first define the basic COM design principles and architectural concepts. In doing so, we will examine the specific problems that COM is meant to solve, and how COM provides solutions for these problems.

The Component Software Problem

The most fundamental problem that COM solves is: How can a system be designed so that binary executables from different vendors, written in different parts of the world and at different times, are able to interoperate? To solve this problem, we must first find answers to these four questions:

• Basic interoperability. How can developers create their own unique binary components, yet be assured that these binary components will interoperate with other binary components built by different developers?
• Versioning. How can one system component be upgraded without requiring all the system components to be upgraded?
• Language independence. How can components written in different languages communicate?
• Transparent cross-process interoperability. How can we give developers the flexibility to write components to run in-process or cross-process, and even cross-network, using one simple programming model?

Additionally, high performance is a requirement for a component software architecture. Although cross-process and cross-network transparency is a laudable goal, it is critical for the commercial success of a binary component marketplace that components interacting within the same address space be able to use each other's services without any undue "system" overhead. Otherwise, the components will not realistically be scalable down to very small, lightweight pieces of software equivalent to C++ classes or graphical user interface (GUI) controls.

COM Fundamentals

The Component Object Model defines several fundamental concepts that provide the model's structural underpinnings. These include:
New features include:

- A binary standard for function calling between components.
- A provision for strongly-typed groupings of functions into interfaces.
- A base interface providing:
  - A way for components to dynamically discover the interfaces implemented by other components.
  - Reference counting to allow components to track their own lifetime and delete themselves when appropriate.
  - A mechanism to identify components and their interfaces uniquely, worldwide.
  - A "component loader" to set up component interactions and, additionally (in the cross-process and cross-network cases), to help manage component interactions.

**Binary Standard**

For any given platform (hardware and operating system combination), COM defines a standard way to lay out virtual function tables (vtables) in memory, and a standard way to call functions through the vtables. Thus, any language that can call functions via pointers (C, C++, Smalltalk, Ada, and even BASIC) all can be used to write components that can interoperate with other components written to the same binary standard. Indirection (the client holds a pointer to a vtable) allows for vtable sharing among multiple instances of the same object class. On a system with hundreds of object instances, vtable sharing can reduce memory requirements considerably, because additional vtables pointing to the same component instance consume much less memory than multiple instances of the same component.

![Virtual Function Tables (VTBL)](image)

**Figure 2. Virtual function tables (VTBL)**

**Objects and Components**

The word *object* tends to mean something different to everyone. To clarify: In COM, an object is a piece of compiled code that provides some service to the rest of the system. To avoid confusion, it is probably best to refer to an object used in COM as a *COM component* or simply as a *component*. This avoids confusing COM components with source-code OOP objects such as those defined in C++.  

---

3 of 22 2/2/10 9:37 PM
COM components support a base interface called **IUnknown** (described later), along with a combination of other interfaces, depending on what functionality the COM component chooses to expose.

COM components usually have some associated data, but unlike C++ objects, a given COM component will never have direct access to another COM component in its entirety. Instead, COM components **always** access other COM components through **interface pointers**. This is a primary architectural feature of the Component Object Model, because it allows COM to completely preserve encapsulation of data and processing, a fundamental requirement of a true component software standard. It also allows for transparent remoting (cross-process or cross-network calling) because all data access is through methods that can be accessed through a proxy-stub pair that forward the request from the client component to the server component and also send back the response.

**Interfaces**

In COM, applications interact with each other and with the system through collections of functions called **interfaces**. Note that all OLE services are simply COM interfaces. A COM "interface" is a strongly-typed **contract** between software components to provide a small but useful set of semantically related operations (methods). An interface is the definition of an expected behavior and expected responsibilities. OLE's drag-and-drop support is a good example. All of the functionality that a component must implement to be a drop target is collected into the **IDropTarget** interface; all the drag source functionality is in the **IDragSource** interface.

Interface names begin with "I" by convention. OLE provides a number of useful general-purpose interfaces (which generally begin with "IOle"), but because anyone can define custom interfaces as well, developers can develop their own interfaces as they deploy component-based applications. Incidentally, a pointer to a COM component is really a pointer to one of the interfaces that the COM component implements; this means that you can only use a COM component pointer to call a method, and **not** to modify data, as described above. Here is an example of an interface, **ILookup**, with two member methods:

```cpp
interface ILookup : public IUnknown
{
    public:
    virtual HRESULT __stdcall LookupByName
        ( LPTSTR lpName, TCHAR **lplpNumber) = 0;
    virtual HRESULT __stdcall LookupByNumber
        ( LPTSTR lpNumber, TCHAR **lplpName) = 0;
};
```

**Attributes of Interfaces**

Given that an interface is a contractual way for a COM component to expose its services, there are several very important points to understand:

- **An interface is not a class.** Although an **instance of a class can be created** (instantiated) to form a COM component, an interface cannot be instantiated by itself because it carries no implementation. A COM component must implement that interface and that COM component must be instantiated in order for an interface to exist. Furthermore, different COM component classes may implement an interface differently, so long as the behavior conforms to the interface definition for each associated component (such as two COM components that implement **IStack** where one uses an array and the other a linked list). Thus the basic
principle of polymorphism fully applies to COM components.

- **An interface is not a COM component.** An interface is just a related group of functions and is the binary standard through which clients and COM components communicate. The COM component can be implemented in any language with any internal state representation, so long as it can provide pointers to interface member functions.

- **COM clients only interact with pointers to interfaces.** When a COM client has access to a COM component, it has nothing more than a **pointer through which it can access the functions in the interface**, called simply an **interface pointer**. The pointer is said to be **opaque** because it hides all aspects of internal implementation. You cannot see the COM component's data, as opposed to C++ **object pointers** through which a client may directly access the object's data. In COM, the client can only call methods of the interface to which it has a pointer. This encapsulation is what allows COM to provide the efficient, safe, robust binary standard that enables local or remote transparency.

- **COM components can implement multiple interfaces.** A COM component can--and typically does--implement more than one interface. That is, the COM class has more than one set of services to provide. For example, a COM class might support the ability to exchange data with COM clients, as well as the ability to save its persistent state information (the data it would need to reload to return to its current state) in a file at the client's request. Each of these abilities is expressed through a different interface (**IDataObject** and **IPersistFile**), so the COM component must implement two interfaces.

- **Interfaces are strongly typed.** Every interface has its own interface identifier (a GUID, which is described later), thereby eliminating any chance of collision that would occur with human-readable names. The difference between components and interfaces has two important implications. When you create a new interface, you must also create a new identifier for that interface. When you use an interface, you must use the identifier for the interface to request a pointer to the interface. This explicit identification improves robustness by eliminating naming conflicts that would result in run-time failure.

- **Interfaces are immutable.** COM interfaces are never versioned, which means that version conflicts between new and old components are avoided. A new version of an interface, created by adding more functions or changing semantics, is an entirely new interface and is assigned a new, unique identifier. Therefore a new interface does not conflict with an old interface even if all that changed is one operation or the semantics (but not even the syntax) of an existing method. Note that, as an implementation efficiency, it is likely that two very similar interfaces can share a common internal implementation. For example, if a new interface adds only one method to an existing interface, and you as the component author wish to support both old-style and new-style clients, you would express both collections of capabilities through two interfaces, yet internally implement the old interfaces as a proper subset of the implementation of the new.

It is convenient to adopt a standard pictorial representation for COM components and their interfaces. The current convention is to draw each interface on a COM component as a "plug-in jack."
The unique use of interfaces in COM provides five major benefits:

1. **The ability for functionality in applications (clients or servers of COM components) to evolve over time.** This is accomplished through a request called `QueryInterface`, which absolutely all COM components support (or else they are not COM components). `QueryInterface` allows a COM component to make more interfaces available to new clients (that is, support new groups of functions) while at the same time retaining complete binary compatibility with existing client code.

In other words, revising a COM component by adding new functionality will not require any recompilation of any existing clients of that component. This is a key solution to the problem of versioning, and is a fundamental requirement for achieving a component software market. COM additionally provides for robust versioning because COM interfaces are immutable, and COM components continue to support old interfaces even while adding new functionality through additional interfaces. This guarantees backwards compatibility as components are upgraded.

Other proposed system object models generally allow developers to change existing interfaces, leading ultimately to versioning problems as components are upgraded. This freedom in other object
models to change interfaces may appear on the surface to handle versioning, but in practice it does not work. For example, if version-checking is done only at object creation time, subsequent users of an instantiated object can easily fail because the object is of the right type but the wrong version. (Per-call version-checking is too expensive to even contemplate!)

2. **Fast and simple object interaction.** Once a client establishes a connection to a COM component, calls to that COM component's services (interface functions) are simply indirect functions calls through two memory pointers. As a result, the performance overhead of interacting with an in-process COM component (an COM component that is in the same address space) as the calling code is negligible.

Calls between COM components in the same process are only a handful of processor instructions slower than a standard direct function call and no slower than a compile-time bound C++ object invocation. In addition, using multiple interfaces per object is efficient because the cost of negotiating interfaces (via `QueryInterface`) is done in groups of functions instead of one function at a time.

3. **Interface reuse.** Design experience suggests that there are many sets of operations that are useful across a broad range of components. For example, it is commonly useful to provide or use a set of functions for reading or writing streams of bytes. In COM, components can reuse an existing interface (such as `IStream`) in a variety of areas. This not only allows for code reuse, but by reusing interfaces, the programmer learns the interface once and can apply it throughout many different applications.

4. **"Local/Remote Transparency."** The binary standard allows COM to intercept an interface call to an object and make instead a remote procedure call (RPC) to an object that is running in another process or on another machine. A key point is that the caller makes this call exactly as it would for an object in the same process. The binary standard enables COM to perform inter-process and cross-network function calls transparently.

Although there is, of course, more overhead in making a remote procedure call, no special code is necessary in the client to differentiate an in-process object from an out-of-process object. This means that as long as the client is written from the start to handle RPC exceptions, all objects (in-process, cross-process, and remote) are available to clients in a uniform, transparent fashion.

Microsoft will be providing a distributed version of COM that will require no modification to existing components in order to gain distributed capabilities. In other words, programmers are completely isolated from networking issues, and components shipped today will operate in a distributed fashion when this future version of COM is released.

5. **Programming language independence.** Any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers can create and use COM components. COM components can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Again, this is because COM, unlike an object-oriented programming language, represents a binary object standard, not a source code standard.

**Globally Unique Identifiers (GUIDs)**

COM uses globally unique identifiers (GUIDs), which are 128-bit integers that are guaranteed to be unique in the universe across space and time, to identify every interface and every COM component
class. These globally unique identifiers are UUIDs (universally unique IDs) as defined by the Open Software Foundation's Distributed Computing Environment. Human-readable names are assigned only for convenience and are locally scoped. This helps ensure that COM components do not accidentally connect to the "wrong" component, interface, or method, even in networks with millions of COM components.

CLSIDs are GUIDs that refer to COM component classes, and IIDs are GUIDs that refer to interfaces. Microsoft supplies a tool (uuidgen) that automatically generates GUIDs. Additionally, the CoCreateGuid function is part of the COM application programming interface (API). Thus, developers create their own GUIDs when they develop COM components and custom interfaces. Through the use of defines, developers don't need to be exposed to the actual 128-bit GUID. For those who want to see real GUIDs in all their glory, the following example shows two GUIDs.

```
DEFINE_GUID(CLSID_PHONEBOOK, 0xc4910d70, 0xba7d, 0x11cd, 0x94, 0xe8,
0x08, 0x00, 0x17, 0x01, 0xa8, 0xa3);

DEFINE_GUID(IID_ILOOKUP, 0xc4910d71, 0xba7d, 0x11cd, 0x94, 0xe8,
0x08, 0x00, 0x17, 0x01, 0xa8, 0xa3);
```

The GUIDs are embedded in the component binary itself and are used by the COM system dynamically at bind time to ensure that no false connections are made between components.

**IUnknown**

COM defines one special interface, IUnknown, to implement some essential functionality. All COM components are required to implement the IUnknown interface and, conveniently, all other COM and OLE interfaces are derived from IUnknown. IUnknown has three methods: QueryInterface, AddRef, and Release. In C++ syntax, IUnknown looks like this:

```
interface IUnknown
{
    virtual HRESULT QueryInterface(IID& iid, void** ppvObj) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
}
```

Figure 6 is a graphical representation of IUnknown.
AddRef and Release are simple reference counting methods. A COM component's AddRef method is called when another COM component is using the interface; the COM component's Release method is called when the other component no longer requires use of that interface. While the COM component's reference count is non-zero it must remain in memory; when the reference count becomes zero, the COM component can safely unload itself, because no other components hold references to it.

QueryInterface is the mechanism that allows clients to dynamically discover (at run time) whether or not an interface is supported by a COM component; at the same time, it is the mechanism that a client uses to get an interface pointer from a COM component. When an application wants to use some function of a COM component, it calls that object's QueryInterface, requesting a pointer to the interface that implements the desired function. If the COM component supports that interface, it will return the appropriate interface pointer and a success code. If the COM component doesn't support the requested interface, it will return an error value. The application will then examine the return code; if successful, it will use the interface pointer to access the desired method. If the QueryInterface failed, the application will take some other action, letting the user know that the desired method is not available.

The following example shows a call to QueryInterface on the PhoneBook component. We are asking this component, "Do you support the ILookup interface?" If the call returns successfully, we know that the COM component supports the ILookup interface and we have a pointer to use to call methods contained in the ILookup interface (either LookupByName or LookupByNumber). If not, we know that the PhoneBook COM component does not implement the ILookup interface.

LPLOOKUP *pLookup;
TCHAR szNumber[64];
HRESULT hRes;
Call QueryInterface on the COM Component PhoneBook, asking for a pointer to the Ilookup interface identified by a unique interface ID.

```c
hRes = pPhoneBook->QueryInterface( IID_ILOOKUP, &pLookup);
if( SUCCEEDED( hRes ) )
{
    pLookup->LookupByName("Daffy Duck", &szNumber);   // Use Ilookup interface pointer.
    pLookup->Release();      // Finished using the IPhoneBook interface pointer.
}
else
{
    // Failed to acquire Ilookup interface pointer.
}
```

Note that AddRef() is not explicitly called in this case because the QueryInterface implementation increments the reference count before it returns an interface pointer.

**COM Library**

The COM Library is a system component that provides the mechanics of COM. The COM Library provides the ability to make IUnknown calls across processes; it also encapsulates all the "legwork" associated with launching components and establishing connections between components. Typically, when an application creates a COM component, it passes the CLSID of that COM component class to the COM Library. The COM Library uses that CLSID to look up the associated server code in the registration database. If the server is an executable, COM launches the EXE and waits for it to register its class factory through a call to CoRegisterClassFactory. (A class factory is the mechanism in COM used to instantiate new COM components.) If that code happens to be a DLL, COM loads the DLL and calls DllGetClassFactory.

COM uses the object's IClassFactory to ask the class factory to create an instance of the COM component, and sends a pointer to the requested interface back to the calling application. The calling application neither knows nor cares where the server application is run; it just uses the returned interface pointer to communicate with the newly created COM component. The COM Library is implemented in COMPOBJ.DLL on Windows, and OLE32.DLL on Windows 95 and Windows NT.

**Interfaces Summary**

To summarize, COM defines several basic fundamentals that provide the underpinnings of the object model. The binary standard allows components written in different languages to call each other's functions. Interfaces are logical groups of related functions--functions that together provide some well-defined capability. IUnknown is the interface that COM defines to allow components to control their own lifespan and to dynamically determine another component's capabilities.

A COM component implements IUnknown to control its lifespan and to provide access to the interfaces it supports. A COM component does not provide direct access to its data. GUIDs provide a unique identifier for each class and interface, thereby preventing naming conflicts. And finally, the COM Library is implemented as part of the operating system, and provides the "legwork" associated with finding and launching COM components.

Now that we have a good understanding of COM's fundamental pieces, let's look at how these pieces fit together to enable component software.
COM Solves the Component Software Problem

COM addresses the four basic problems associated with component software:

- Basic component interoperability
- Versioning
- Language independence
- Transparent cross-process interoperability

Additionally, COM provides a high-performance architecture to meet the requirements of a commercial component market.

Basic Interoperability and Performance

These are provided by COM's use of vtables to define a binary interface standard for method calling between components. Calls between COM components in the same process are only a handful of processor instructions slower than a standard direct function call and no slower than a compile-time bound C++ object invocation.

Versioning

A good versioning mechanism allows one system component to be updated without requiring updates to all the other components in the system. Versioning in COM is implemented using interfaces and IUnknown::QueryInterface. The COM design completely eliminates the need for things like version repositories or central management of component versions.

When a software module is updated, it is generally to add new functionality, or to improve existing functionality. In COM, you add new functionality to your COM component by adding support for new interfaces. Because the existing interfaces don't change, other components that rely on those interfaces continue to work. Newer components that know about the new interfaces can use those newly exposed interfaces.

Because QueryInterface calls are made at run time without any expensive call to some "capabilities database" (as used in some other system object models), the current capabilities of a COM component can be efficiently evaluated each time the component is used. When new features become available, applications that know how to use them will begin to do so immediately.

Improving existing functionality is even easier. Because the syntax and semantics of an interface remain constant, you are free to change the implementation of an interface, without breaking other developers' components that rely on the interface. For example, say you have a component that supports the (hypothetical) IStack interface, which (hypothetically) includes methods like Push and Pop. You've currently implemented the interface as an array, but you decide that a linked list would be more appropriate. Because the methods and parameters do not change, you can freely replace the old implementation with a new one, and applications that use your component will get the improved linked list functionality "for free."

Windows and OLE use this technique to provide improved system support. For example, in OLE today, Structured Storage is implemented as a set of interfaces that currently use the C run-time file I/O functions internally. In the next major release of the Windows NT operating system, those same
interfaces will write directly to the file system. The syntax and semantics of the interfaces remain constant; only the implementation changes. Existing applications will be able to use the new implementation without any changes; they will get the improved functionality "for free."

The combination of the use of interfaces (immutable, well-defined "functionality sets" that are extruded by components) and QueryInterface (the ability to cheaply determine at run time the capabilities of a specific COM component) enables COM to provide an architecture in which components can be dynamically updated, without requiring updates to other reliant components. This is a fundamental strength of COM over other proposed object models.

COM solves the versioning/evolution problem where the functionality of objects can change independently of clients of that object without rendering existing clients incompatible. In other words, COM defines a system in which components continue to support the interfaces through which they provided services to older clients, as well as support new and better interfaces through which they can provide services to newer clients. At run time, old and new clients can safely coexist with a given COM component. Errors can only occur at easily handled times: bind time or during a QueryInterface call. There is no chance for random crashes, such as those that occur when an expected method on an object simply does not exist, or its parameters have changed.

**Language Independence**

Components can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Again, this is because COM, unlike an object-oriented programming (OOP) language, represents a binary object standard, not a source code standard. This is a fundamental benefit of a component software architecture over object-oriented programming languages. Objects defined in an OOP language typically interact only with other objects defined in the same language. This necessarily limits their reuse. At the same time, an OOP language can be used in building COM components, so the two technologies are actually quite complementary. COM can be used to "package" and further encapsulate OOP objects into components for widespread reuse, even within very different programming languages.

**Transparent Cross-Process Interoperability**

It would be relatively easy to address the problem of providing a component software architecture if software developers could assume that all interactions between components occurred within the same process space. In fact, other proposed system object models do make this basic assumption.

The bulk of the work in defining a true component software model involves the transparent bridging of process barriers. In the design of COM, it was understood from the beginning that interoperability had to occur across process spaces because most applications could not be expected to be rewritten as dynamic-link libraries (DLLs) loaded into shared memory. Also, by solving the problem of cross-process interoperability, COM solves the problem of components communicating transparently between different computers across a network, using exactly the same programming interface used for components communicating on the same computer.

The COM Library is the key to providing transparent cross-process interoperability. As discussed in the last section, the COM Library encapsulates all the "legwork" associated with finding and launching components and managing the communication between components. As shown earlier, the COM Library insulates components from the location differences. This means that COM components can interoperate freely with other COM components running in the same process, in a different
process, or across the network. The code needed to implement or use a COM component in any of those cases is exactly the same. Thus, when a new COM Library is released with support for cross-network interaction, existing COM components will be able to work in a distributed fashion without requiring source-code changes, recompilation, or redistribution to customers.

**Local and Remote Transparency**

COM is designed to allow clients to *transparently* communicate with components regardless of where those components are running, be it the same process, the same machine, or a different machine. This means that there is a *single programming model* for all types of COM components, not only for clients of those COM components but also for the servers of those COM components.

From a client's point of view, all COM components are accessed through interface pointers. A pointer must be in-process, and in fact, any call to an interface function always reaches *some* piece of in-process code first. If the COM component is in-process, the call reaches it directly. If the COM component is out-of-process, the call first reaches what is called a *proxy* object provided by COM itself that generates the appropriate remote procedure call to the other process or the other machine. Note that the client should be programmed from the start to handle RPC exceptions; then it can transparently connect to an object that is in-process, cross-process, or remote.

From a server's point of view, all calls to a COM component's interface functions are made through a pointer to that interface. Again, a pointer only has context in a single process, and so the caller must always be some piece of in-process code. If the COM component is in-process, the caller is the client itself. Otherwise, the caller is a *stub* object provided by COM that picks up the remote procedure call from the proxy in the client process and turns it into an interface call to the server COM component.

As far as both clients and servers know, they always communicate directly with some other in-process code, as illustrated in Figure 7.
Figure 7. Clients always call in-process code; COM components are always called by in-process code. COM provides the underlying transparent RPC.

The bottom line is that dealing with local or remote COM components is transparent and identical to dealing with in-process COM components. This local/remote transparency has a number of key benefits:

- **A common solution to problems that are independent of the distance between client and server**: For example, connection, function invocation, interface negotiation, feature evolution, and so forth, occur the same way for components interoperating in the same process as for components interoperating across global networks.

- **Programmers leverage their learning.** New services are simply exposed through new interfaces, and once programmers learn how to deal with interfaces, they already know how to deal with new services that will be created in the future. This is a great improvement over environments in which each service is exposed in a completely different fashion. For example, Microsoft is working with other ISVs to extend OLE services. These new services, which will be quite diverse in function, will all be very similar in their implementations because they will simply be sets of COM interfaces.

- **Systems implementation is centralized.** The implementors of COM can focus on making the central process of providing this transparency as efficient and powerful as possible,
benefitting every piece of code that uses COM.

- **Interface designers concentrate on design.** In designing a suite of interfaces, the designers can spend their time in the essence of the design—the contracts between the parties—without having to think about the underlying communication mechanisms for any interoperability scenario. COM provides those mechanisms for free, including network transparency.

## COM and the Client/Server Model

### Clients, Servers, and Object Implementors

The interaction between COM components and the users of those COM components in COM is in one sense based on a client/server model. We have already used the term *client* to refer to some piece of code that is using the services of a COM component. Because a COM component supplies services, the implementor of that component is usually called the *server*—the COM component that serves those capabilities. A client/server architecture in any computing environment leads to greater robustness: If a server process crashes or is otherwise disconnected from a client, the client can handle that problem gracefully and even restart the server if necessary.

Because robustness is a primary goal in COM, a client/server model naturally fits. Because COM allows clients and servers to exist in different process spaces (as desired by component providers), crash protection can be provided between the different components making up an application. For example, if one component in a component-based application fails, the entire application will not crash. In contrast, object models that are only in-process cannot provide this same fault tolerance. The ability to cleanly separate object clients and object servers in different process spaces is very important for a component software standard that promises to support sophisticated applications.

Unlike other object models we know of, COM is unique in allowing clients to also represent themselves as servers. In fact many interesting designs have two (or more) components using interface pointers on each other, thus becoming clients and servers simultaneously. In this sense, COM also supports the notion of peer-to-peer computing, and so is quite different—and, we think, more flexible and useful—from other proposed object models in which clients *never* represent themselves as objects.

### Servers: In-Process and Out-Of-Process

In general a server is some piece of code that implements some COM component such that the COM Library and its services can run that code and have it create COM components.

Any specific server can be implemented in one of a number of flavors, depending on the structure of the code module and its relationship to the client process that will be using it. A server is either *in-process*, which means its code executes in the same process space as the client (as a DLL), or *out-of-process*, which means it runs in another process on the same machine or in another process on a remote machine (as an EXE). These three types of servers are called *in-process, local*, and *remote*.

COM component implementors choose the type of server based on the requirements of implementation and deployment. COM is designed to handle all situations, from those that require the deployment of many small, lightweight in-process components (like OLE Controls, but
conceivably even smaller) up to those that require deployment of huge components, such as a central corporate database server. And as discussed, all COM servers look the same to client applications, whether they are in-process, local, or remote.

**Custom Interfaces and Interface Definitions**

When a developer defines a new custom interface, she or he can create an interface definition using the Interface Description Language (IDL). From this interface definition, the Microsoft IDL compiler generates header files for use by applications using that interface, and source code to create proxy and stub objects that handle remote procedure calls. The IDL used and supplied by Microsoft is based on simple extensions to the OSF DCE IDL, a growing industry standard for RPC-based distributed computing.

IDL is only a tool for the convenience of the interface designer and is not central to COM's interoperability. It simply saves the developer from manually creating header files for each programming environment and from creating proxy and stub objects by hand. Note that IDL is not necessary unless you are defining a custom interface for an object--proxy and stub objects are already provided with the COM Component Library for all COM and OLE interfaces. Here is the IDL file used to define a custom interface, `ILookup`, that is implemented by the PhoneBook object:

```idl
[object,
    //Use the GUID for the ILookup interface.
    uuid(c4910d71-ba7d-11cd-94e8-08001701a8a3),
    pointer_default(unique)
]
interface ILookup : IUnknown  // ILookup interface derives from IUnknown.
{
    import "unknwn.idl";     // Bring in the supplied IUnknown IDL.
    // Define member function LookupByName:
    HRESULT LookupByName( [in] LPTSTR lpName, [out, string] WCHAR **lplpNumber);
    // Define member function LookupByNumber:
    HRESULT LookupByNumber( [in] LPTSTR lpNumber, [out, string] WCHAR **lplpName);
}
```

**COM and Application Structure**

COM is not a specification for how applications are structured, it is a specification for how applications interoperate. For this reason, COM is not concerned with the internal structure of an application. That is the job of the programmer, and also depends on the programming languages and development environments used. Conversely, programming environments have no set standards for working with objects outside the immediate application.

C++, for example, works extremely well with objects inside an application, but has no support for working with objects outside the application. Generally, other programming languages are the same. COM, through language-independent interfaces, picks up where programming languages leave off, providing network-wide interoperability of components making up an integrated application.

**Client/Server Summary**
The core of the Component Object Model is a specification for how components and their clients interact. As a specification it defines a number of other standards for interoperability of software components:

- A binary standard for function calling between components.
- A provision for strongly typed groupings of functions into interfaces.
- A base `IUnknown` interface providing:
  - A way for components to dynamically discover the interfaces supported by other components (`QueryInterface`).
  - Reference counting to encapsulate component lifetime (`AddRef` and `Release`).
  - A mechanism to uniquely identify components and their interfaces (GUIDs).

In addition to being a specification, COM is also an implementation contained in the COM Library. The implementation is provided through a library (such as a DLL on Microsoft Windows, Windows 95, or Windows NT) that includes:

- A small number of fundamental API functions that facilitate the creation of COM applications, both clients and servers. For clients, COM supplies basic COM component creation functions; for servers it supplies facilities to expose their COM components.
- Implementation locator services through which COM determines from a class identifier which server implements that class and where that server is located. This includes support for a level of indirection, usually a system registry, between the identity of an COM component class and the packaging of the implementation such that clients are independent of the packaging (so packaging can change in the future).
- Transparent remote procedure calls when a COM component is running in a local or remote server, as illustrated in Figure 7.

In general, only one vendor needs to, or should, implement a COM Library for any particular operating system. For example, Microsoft is implementing COM on Windows, Windows NT, and the Apple Macintosh. Other vendors are implementing COM on other operating systems, including specific versions of UNIX. Also, it is important to note that COM draws a very clear distinction between:

- The object model and the wire-level protocols for distributed services, which are the same on all platforms

and

- Platform-specific operating system services (for example, local security, network transports, and so on)

Therefore, developers are not constrained to use new and specific models for the services of different operating systems, yet they can develop components that interoperate with components on other platforms.

All in all, only with a binary standard on a given platform and a wire-level protocol for cross-machine component interaction can an object model provide the type of structure necessary for full...
interoperability between all applications and between all different machines in a network. With a binary and network standard, COM opens the doors for a revolution in innovation without a revolution in programming or programming tools.

Appendix 1. The Problem with Implementation Inheritance

Implementation inheritance--the ability of one component to "subclass" or inherit some of its functionality from another component--is a very useful technology for building applications. Implementation inheritance, however, can create many problems in a distributed, evolving object system.

The problem with implementation inheritance is that the "contract" or relationship between components in an implementation hierarchy is not clearly defined; it is implicit and ambiguous. When the parent or child component changes its behavior unexpectedly, the behavior of related components may become undefined. This is not a problem when the implementation hierarchy is under the control of a defined group of programmers who can make updates to all components simultaneously. But it is precisely this ability to control and change a set of related components simultaneously that differentiates an application, even a complex application, from a true distributed object system.

So although implementation inheritance can be a very good thing for building applications, it is not appropriate for a system object model that defines an architecture for component software.

In a system built of components provided by a variety of vendors, it is critical that a given component provider be able to revise, update, and distribute (or redistribute) a product without breaking existing code in the field that is using the previous revision or revisions of that component. In order to achieve this, it is necessary that the actual interface on the component used by such clients be crystal clear to both parties. Otherwise, how can the component provider be sure to maintain that interface, and thus not break the existing clients?

From observation, the problem with implementation inheritance is that it is significantly easier for programmers not to be clear about the actual interface between a base and derived class than it is to be clear. This usually leads implementors of derived classes to require source code to the base classes; in fact, most application framework development environments that are based on inheritance provide full source code for exactly this reason.

The bottom line is that inheritance, although very powerful for managing source code in a project, is not suitable for creating a component-based system where the goal is for components to reuse each others' implementations without knowing any internal structures of the other objects. Inheritance violates the principle of encapsulation, the most important aspect of an object-oriented system.

An Example of the Implementation Inheritance Problem

The following C++ example illustrates the technical heart of the robustness problem:

```cpp
class CBase
{
public:
void DoSomething(void) { ... if (condition) this->Sample(); ...}
virtual void Sample(void);
};
```
This is the classic paradigm of reuse in implementation inheritance: A base class periodically makes calls to its own virtual functions, which may be overridden by its derived classes. In practice, in such a situation `CDerived` can become, and therefore often will become, intimately dependent on `exactly when and under what conditions Sample will be invoked by the class CBase`.

If, at present, all such `Sample` invocations by `CBase` are intended (long-term) as hooks for the derived class, there is no problem. There are two cases, however; either the implementation of `CBase::Sample` is implemented as

```cpp
void CBase::Sample(void)
{
    //Do absolutely nothing.
}
```

or it is not empty.

If the implementation of `CBase::Sample` is `not empty`, it is carrying out some useful and likely needed transformation on the internal state of `CBase`. Thus, it is questionable whether `all of the invocations of Sample` are for the support of derived classes; some of them instead are likely to be only for the purpose of carrying out this transformation. That transformation is part of current implementation of `CBase`.

Thus, in summary, `CDerived` becomes coupled to details of that current implementation of `CBase`; the interface between the two is not clear and precise.

Further coupling comes from the fact that the implementation of `CDerived::Sample`, in addition to performing its own role, must be sure to carry out the transformation carried out in `CBase::Sample`. It can do this by reimplementing the code in `CBase::CBase` itself, but this causes obvious coupling problems. Alternatively, it can itself invoke the `CBase::Sample` method:

```cpp
void CDerived::Sample(void)
{
    [Do some work]
    ...
    CBase::Sample();
    ...
    [Do other work]
}
```

However, it is very unclear what is appropriate or possible for `CDerived` to do in the areas marked "Do some work" and "Do other work." What is appropriate depends, again, heavily on the current implementation of `CBase::Sample`.

If, in contrast, `CBase::Sample` is empty, we likely are not in immediate danger of surreptitious coupling. In the implementation of `CBase`, invoking `Sample` clearly serves no immediately useful purpose, and so it is likely that indeed all invocations of `Sample` in `CBase` are only for the support of `CDerived`. Consider, however, the case in which the `CDerived` class is reused:
class CAnother : public CDerived
{
public:
    virtual void Sample(void);
};

Though `CBase::Sample` had a trivial implementation, `CDerived::Sample` will not (why override it if otherwise?). The relationship of `CAnother` to `CDerived` thus becomes as problematic as the `CDerived-CBase` relationship in the previous case.

This is the architectural heart of the problem observed in practice that leads to a view that implementation inheritance is unacceptable as the mechanism by which independently developed binary components are reused and refined.

COM provides two other mechanisms for code reuse, called containment/delegation and aggregation. Both of these reuse mechanisms allow objects to exploit existing implementation while avoiding the problems of implementation inheritance. See Appendix 2 of this article for an overview of these alternate reuse mechanisms.

**Appendix 2. COM Reusability Mechanisms**

The key point to building reusable components is black-box reuse, which means the piece of code attempting to reuse another component knows nothing, and needs to know nothing, about the internal structure or implementation of the component being used. In other words, the code attempting to reuse a component depends upon the behavior of the component and not the exact implementation. As illustrated in Appendix 1, implementation inheritance does not achieve black-box reuse.

To achieve black-box reusability, COM supports two mechanisms through which one COM component may reuse another. For convenience, the object being reused is called the **inner object** and the object making use of that inner object is the **outer object**.

- **Containment/Delegation.** The outer object behaves like an object client to the inner object. The outer object "contains" the inner object, and when the outer object wishes to use the services of the inner object, the outer object simply delegates implementation to the inner object's interfaces. In other words, the outer object uses the inner object's services to implement some (or possibly all) of its own functionality.

- **Aggregation.** The outer object wishes to expose interfaces from the inner object as if they were implemented on the outer object itself. This is useful when the outer object would always delegate every call to one of its interfaces to the same interface of the inner object. Aggregation is a convenience to allow the outer object to avoid extra implementation overhead in such cases.

These two mechanisms are illustrated in Figures 8 and 9. The important part of both these mechanisms is how the outer object appears to its clients. As far as the clients are concerned, both objects implement interfaces A, B, and C. Furthermore, the client treats the outer object as a black box, and thus does not care, nor does it need to care, about the internal structure of the outer object--the client only cares about behavior.
Figure 8. Containment of an inner object and delegation to its interfaces

Aggregation is almost as simple to implement. The trick here is for COM to preserve the function of QueryInterface for COM component clients even as an object exposes another COM component's interfaces as its own. The solution is for the inner object to delegate IUnknown calls in its own interfaces, but also allow the outer object to access the inner object's IUnknown functions directly. COM provides specific support for this solution.
Figure 9. Aggregation of an inner object where the outer object exposes one or more of the inner object's interfaces as its own.