0Part II: Component Object Model Programming Interface

Part II contains the programming interface to COM, the suite of interfaces and APIs by which Component Object Model software is implemented and used.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

1Component Object Model Technical Overview

Chapter 1 introduced some important challenges and problems in computing today and the Component Object Model as a solution to these problems. Chapter 1 introduced interfaces, mentioned the base interface called IUnknown, and described how interfaces are generally used to communicate between an object and a client of that object, and explained the role that COM has in that communication to provide location transparency.

Yet there are plenty of topics that have not been covered in much technical detail, specifically, how certain mechanisms work, some of the interfaces involved, and how some of these interfaces are used on a high level. This chapter will describe COM in a more technical light but not going as far as describing individual interface functions or COM Library API functions. Instead, this chapter will refer to later chapters in the COM Specification that cover various topics in complete detail including the specifications for functions and interfaces themselves.

This chapter is generally organized in the same order as Chapter 1 and covers the following topics which are then treated in complete detail in the indicated chapters:

- Objects and Interfaces: A comparison of interfaces to C++ classes, the IUnknown interface (including the QueryInterface function and reference counting), the structure of an instantiated interface and the benefits of that structure, and how clients of objects deal with interfaces. Chapter 3 covers the underlying interfaces and API functions themselves.
- COM Applications: The responsibilities of all applications making use of COM which includes rules for memory management. How applications meet these responsibilities is covered in Chapter 4.
- COM Clients and Servers: The roles and responsibilities of each specific type of application, the use of class identifiers, and the COM Library's role in providing communication. Chapter 5 and 6 treat COM Clients and Servers separately. How COM achieves location transparency is described in Chapter 7.

This page intentionally left blank.

- Reusability: A discussion about why implementation inheritance is not used in COM and what mechanisms are instead available. How an object server is written to handle the COM mechanisms is a topic of Chapter 6.
- Connectable Objects: A brief overview of the connection point interfaces and semantics. The actual functional specification of connectable objects is in Chapter 9.
- Persistent Storage: A detailed look at what persistent storage is, what benefits it holds for applications including incremental access and transactioning support, leaving the APIs and interface specifications to Chapter 10.
- Persistent, Intelligent Names: Why it is important to assign names to individual object instantiations (as opposed to a class identifier for an object class) and the mechanisms for such naming including moniker objects. The interfaces a moniker implements as well as other support functions are described in Chapter 11.
- Uniform Data Transfer: The separation of transfer protocols from data exchange, improvements to data format descriptions, the expansion of available exchange mediums (over global memory), and data change notification mechanisms. New data structures and interfaces specified to support data transfer is given in Chapter 12.

1.1Objects and Interfaces

Chapter 1 described that interfaces are—strongly typed semantic contracts between client and object—and that an object in COM is any structure that exposes its functionality through the interface mechanism. In addition, Chapter 1 noted how interfaces follow a binary standard and how such a standard enables clients and objects to interoperate regardless of the programming languages used to implement them. While the *type* of an interface is by colloquial convention referred to with a name starting with an "I" (for interface), this name is only of significance in source-level programming tools. Each interface itself—the

Copyright © 1995 Microsoft Corporation
All Rights Reserved

immutable contract, that is—as a functional group is referred to at runtime with a globally-unique interface identifier, an "IID" that allows a client to ask an object if it supports the semantics of the interface without unnecessary overhead and without versioning problems. Clients ask questions using a QueryInterface function that all objects support through the base interface, IUnknown.

Furthermore, clients always deal with objects through interface pointers and never directly access the object itself. Therefore an interface is not an object, and an object can, in fact, have more than one interface if it has more than one group of functionality it supports.

Let's now turn to how interfaces manifest themselves and how they work.

1.1.1Interfaces and C++ Classes

As just reiterated, an interface is not an object, nor is it an object class. Given an interface definition by itself, that is, the type definition for an interface name that begins with "I," you cannot create an object of that type. This is one reason why the prefix "I" is used instead of the common C++ convention of using a "C" to prefix an object class, such as CMyClass. While you can instantiate an object of a C++ class, you cannot instantiate an object of an interface type.

In C++ applications, interfaces are, in fact, defined as *abstract base classes*. That is, the interface is a C++ class that contains nothing but pure virtual member functions. This means that the interface carries no implementation and only prescribes the function signatures for some other class to implement—C++ compilers will generate compile-time errors for code that attempts to instantiate an abstract base class. C++ applications implement COM objects by inheriting these function signatures from one or more interfaces, overriding each interface function, and providing an implementation of each function. This is how a C++ COM application "implements interfaces" on an object.

Implementing objects and interfaces in other languages is similar in nature, depending on the language. In C, for example, an interface is a structure containing a pointer to a table of function pointers, one for each method in the interface. It is very straightforward to use or to implement a COM object in C, or indeed in any programming language which supports the notion of function pointers. No special tools or language enhancements are required (though of course such things may be desirable).

The abstract-base class comparison exposes an attribute of the "contract" concept of interfaces: if you want to implement any single function in an interface, you must provide some implementation for *every* function in that interface. The implementation might be nothing more than a single return statement when the object has nothing to do in that interface function. In most cases there is some meaningful implementation in each function, but the number of lines of code varies greatly (one line to hundreds, potentially).

A particular object will provide implementations for the functions in every interface that it supports. Objects which have the same set of interfaces and the same implementations for each are often said (loosely) to be instances of the same class because they generally implement those interfaces in a certain way. However, all access to the instances of the class by clients will only be through interfaces; clients know nothing about an object other than it supports certain interfaces. As a result, classes play a much less significant role in COM than they do in other object oriented systems.

COM uses the word "interface" in a sense different from that typically used in object-oriented programming using C++. In the C++ context, "interface" describes *all* the functions that a class supports and that clients of an object can call to interact with it. A COM interface refers to a pre-defined group of related functions that a COM class implements, but does not necessarily represent *all* the functions that the class supports. This separation of an object's functionality into groups is what enables COM and COM applications to avoid the problems inherent with versioning traditional all-inclusive interfaces.

1.1.2Interfaces and Inheritance

COM separates class hierarchy (or indeed any other *implementation* technology) from interface hierarchy and both of those from any implementation hierarchy. Therefore, interface inheritance is only applied to reuse the definition of the contract associated with the base interface. There is no selective inheritance in COM: if one interface inherits from another, it includes all the functions that the other interface defines, for the same reason than an object must implement all interface functions it inherits.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

Inheritance is used sparingly in the COM interfaces. Most of the pre-defined interfaces inherit directly from IUnknown (to receive the fundamental functions like QueryInterface), rather than inheriting from another interface to add more functionality. Because COM interfaces are inherited from IUnknown, they tend to be small and distinct from one another. This keeps functionality in separate groups that can be independently updated from the other interfaces, and can be recombined with other interfaces in semantically useful ways.

In addition, interfaces only use single inheritance, never multiple inheritance, to obtain functions from a base interface. Providing otherwise would significantly complicate the interface method call sequence, which is just an indirect function call, and, further, the utility of multiple inheritance is subsumed within the capabilities provided by QueryInterface.

1.1.3Interface Definitions: IDL

When a designer creates an interface, that designer usually defines it using an Interface Description Language (IDL). From this definition an IDL compiler can generate header files for programming languages such that applications can use that interface, create proxy and stub objects to provide for remote procedure calls, and output necessary to enable RPC calls across a network.

IDL is simply a tool (one of possibly many) for the convenience of the interface designer and is not central to COM's interoperability. It really just saves the designer from manually creating many header files for each programming environment and from creating proxy and stub objects by hand, which would not likely be a fun task.

Chapter 13 describes the Microsoft Interface Description Language in detail. In addition, Chapter 14 covers Type Libraries which are the machine readable form of IDL, used by tools and other components at runtime.

1.1.4Basic Operations: The IUnknown *Interface*

All objects in COM, through any interface, allow clients access to two basic operations:

- Navigating between multiple interfaces on an object through the QueryInterface function.
- Controlling the object's lifetime through a reference counting mechanism handled with functions called AddRef and Release.

Both of these operations as well as the three functions (and only these three) make up the IUnknown interface from which all other interfaces inherit. That is, all interfaces are polymorphic with IUnknown so they all contain QueryInterface, AddRef, and Release functions.

Navigating Multiple Interfaces: the *QueryInterface* Function

As described in Chapter 1, QueryInterface is the mechanism by which a client, having obtained one interface pointer on a particular object, can request additional pointers to *other* interfaces on that same object. An input parameter to QueryInterface is the interface identifier (IID) of the interface being requested. If the object supports this interface, it returns that interface on itself through an accompanying output parameter typed as a generic void; if not, the object returns an error.

In effect, what QueryInterface accomplishes is a switch between contracts on the object. A given interface embodies the interaction that a certain contract requires. Interfaces are groups of functions because contracts in practice invariably require more than one supporting function. QueryInterface separates the request "Do you support a given contract?" from the high-performance use of that contract once negotiations have been successful. Thus, the (minimal) cost of the contract negotiation is amortized over the subsequent use of the contract.

Conversely, QueryInterface provides a robust and reliable way for a component to indicate that in fact does *not* support a given contract. That is, if using QueryInterface one asks an "old" object whether it supports a "new" interface (one, say, that was invented after the old object has been shipped), then the old object will reliably and robustly answer "no;" the technology which supports this is the algorithm by which IIDs are allocated. While this may seem like a small point, it is excruciatingly important to the overall

Copyright © 1995 Microsoft Corporation
All Rights Reserved

architecture of the system, and this capability to robustly inquire of old things about new functionality is, surprisingly, a feature not present in most other object architectures.

The strengths and benefits of the QueryInterface mechanism need not be reiterated here further, but there is one pressing issue: how does a client obtain its first interface pointer to an object? That question is of central interest to COM applications but has no one answer. There are, in fact, four methods through which a client obtains its first interface pointer to a given object:

- Call a COM Library API function that creates an object of a pre-determined type—that is, the function will only return a pointer to one specific interface for a specific object class.
- Call a COM Library API function that can create an object based on a class identifier and that returns any type interface pointer requested.
- Call a member function of some interface that creates another object (or connects to an existing one) and returns an interface pointer on that separate object.[1]
- Implement an object with an interface through which other objects pass their interface pointer to the client directly. This is the case where the client is an object implementor and passes a pointer to its object to another object to establish a bi-directional connection.

Reference Counting: Controlling Object Life-cycle

Just like an application must free memory it allocated once that memory is no longer in use, a client of an object is responsible for freeing the object when that object is no longer needed. In an object-oriented system the client can only do this by giving the object an instruction to free itself.

However, the difficulty lies in having the object know when it is safe to free itself. COM objects, which are dynamically allocated, must allow the client to decide when the object is no longer in use, especially for local or remote objects that may be in use by multiple clients at the same time—the object must wait until *all* clients are finished with it before freeing itself.

COM specifies a *reference counting* mechanism to provide this control. Each object maintains a 32-bit reference count that tracks how many clients are connected to it, that is, how many pointers exist to any of its interfaces in any client. The use of a 32-bit counter (more than four billions clients) means that there's virtually no chance of overloading the count.

The two IUnknown functions of AddRef and Release that all objects must implement control the count: AddRef increments the count and Release decrements it. When the reference count is decremented to zero, Release is allowed to free the object because no one else is using it anywhere. Most objects have only one implementation of these functions (along with QueryInterface) that are shared between all interfaces, though this is just a common implementation approach. Architecturally, from a client's perspective, reference counting is strictly and clearly a per-interface notion.

Whenever a client calls a function that returns a new interface pointer to it, such as QueryInterface, the function being called is responsible for incrementing the reference count through the returned pointer. For example, when a client first creates an object it receives back an interface pointer to an object that, from the client's point of view, has a reference count of one. If the client calls QueryInterface once for another interface pointer, the reference count is two. The client must then call Release through *both* pointers (in any order) to decrement the reference count to zero before the object as a whole can free itself.

In general, every copy of any pointer to any interface requires a reference count on it. Chapter 3, however, identifies some important optimizations that can be made to eliminate extra unnecessary overhead with reference counting and identifies the specific cases in which calling AddRef is absolutely necessary.

1.1.5How an Interface Works

An instantiation of an interface implementation (because the defined interfaces themselves cannot be instantiated without implementation) is simply pointer to an array of pointers to functions. Any code that has access to that array—a pointer through which it can access the array—can call the functions in that

[1] Connecting to objects through an "intelligent/persistent name" (moniker) falls into this category.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

interface. In reality, a pointer to an interface is actually a pointer to a pointer to the table of function pointers. This is an inconvenient way to speak about interfaces, so the term "interface pointer" is used instead to refer to this multiple indirection. Conceptually, then, an interface pointer can be viewed simply as a pointer to a function table in which you can call those functions by dereferencing them by means of the interface pointer as shown in Figure 2-1.

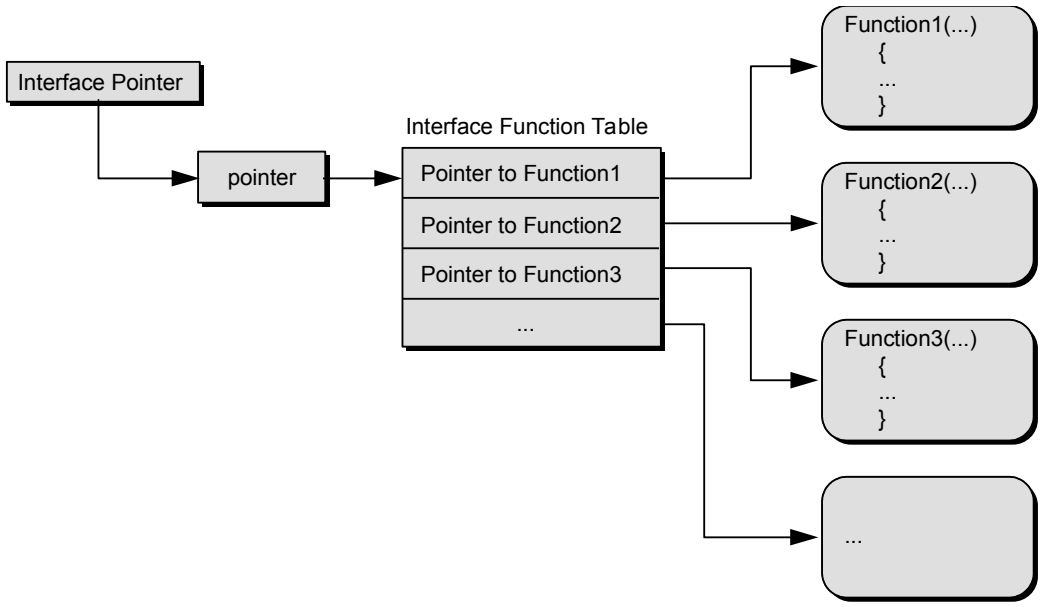
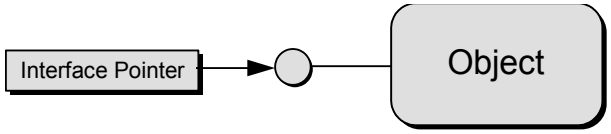


Figure 2-1: An interface pointer is a pointer to a pointer to an array of pointers to the functions in the interface.

Since these function tables are inconvenient to draw they are represented with the "plug-in jack" or "bubbles and push-pins" diagram first shown in Chapter 1 to mean exactly the same thing:



Objects with multiple interfaces are merely capable of providing more than one function table. Function tables can be created manually in a C application or almost automatically with C++ (and other object oriented languages that support COM). Chapter 3 describes exactly how this is accomplished along with how the implementation of the interface functions know exactly which object is being used at any given time.

With appropriate compiler support (which is inherent in C and C++), a client can call an interface function through the name of the function and not its position in the array. The names of functions and the fact that an interface is a type allows the compiler to check the types of parameters and return values of each interface function call. In contrast, such type-checking is not available even in C or C++ if a client used a position-based calling scheme.

1.1.6Interfaces Enable Interoperability

COM is designed around the use of interfaces because interfaces enable interoperability. There are three properties of interfaces that provide this: polymorphism, encapsulation, and transparent remoting.

Polymorphism

Polymorphism means the ability to assume many forms, and in object-oriented programming it describes the ability to have a single statement invoke different functions at different times. All COM interfaces are polymorphic; when you call a function using an interface pointer, you don't specify which implementation is invoked. A call to `pInterface->SomeFunction` can cause different code to run depending on

Copyright © 1995 Microsoft Corporation
All Rights Reserved

what kind of object is the implementor of the interface pointed by pInterface—while the semantics of the function are always the same, the implementation details can vary.

Because the interface standard is a binary standard, clients that know how to use a given interface can interact with any object that supports that interface *no matter how the object implements that contract*. This allows interoperability as you can write an application that can cooperate with other applications without you knowing who or what they are beforehand.

Encapsulation

Other advantages of COM arise from its enforcement of encapsulation. If you have implemented an interface, you can change or update the implementation without affecting any of the clients of your class. Similarly, you are immune to changes that others make in their implementations of their interfaces; if they improve their implementation, you can benefit from it without recompiling your code.

This separation of contract and implementation can also allow you to take advantage of the different implementations underlying an interface, even though the interface remains the same. Different implementations of the same interface are interchangeable, so you can choose from multiple implementations depending on the situation.

Interfaces provides extensibility; a class can support new functionality by implementing additional interfaces without interfering with any of its existing clients. Code using an object's ISomeInterface is unaffected if the class is revised to in addition support IAnotherInterface.

Transparent Remoting

COM interfaces allow one application to interact with others anywhere on the network just as if they were on the same machine. This expands the range of an object's interoperability: your application can use any object that supports a given contract, no matter how the object implements that contract, and no matter what machine the object resides on.

Before COM, class code such as C++ class libraries ran in same process, either linked into the executable or as a dynamic-link library. Now class code can run in a separate process, on the same machine or on a different machine, and your application can use it with no special code. COM can intercept calls to interfaces through the function table and generate remote procedure calls instead.

1.2COM Application Responsibilities

Each process that uses COM in any way—client, server, object implementor—is responsible for three things:

1. Verify that the COM Library is a compatible version with the COM function CoBuildVersion.
2. Initialize the COM Library before using any other functions in it by calling the COM function CoInitialize.
3. Un-initialize the COM Library when it is no longer in use by calling the COM function CoUninitialize.

While these responsibilities and functions are covered in detail in Chapter 4, note first that most COM Library functions, primarily those that deal with the COM foundation, are prefixed with "Co" to identify their origin. The COM Library may implement other functions to support persistent storage, naming, and data transfer without the "Co" prefix.

1.2.1Memory Management Rules

In COM there are many interface member functions and APIs which are called by code written by one programming organization and implemented by code written by another. Many of the parameters and return values of these functions are of types that can be passed around by value; however, sometimes there arises the need to pass data structures for which this is not the case, and for which it is therefore necessary that the caller and the callee agree as to the allocation and de-allocation policy. This could in theory be decided and documented on an individual function by function basis, but it is much more reasonable to adopt a universal convention for dealing with these parameters. Also, having a clear

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

convention is important technically in order that the COM remote procedure call implementation can correctly manage memory.

Memory management of pointers to interfaces is always provided by member functions in the interface in question. For all the COM interfaces these are the AddRef and Release functions found in the IUnknown interface, from which again all other COM interfaces derive (as described earlier in this chapter). This section relates only to non-by-value parameters which are *not* pointers to interfaces but are instead more mundane things like strings, pointers to structures, etc.

The COM Library provides an implementation of a memory allocator (see CoGetMalloc and CoTaskMemAlloc). Whenever ownership of an allocated chunk of memory is passed through a COM interface or between a client and the COM library, this allocator must be used to allocate the memory.[2]

Each parameter to and the return value of a function can be classified into one of three groups: an **in** parameter, an **out** parameter (which includes return values), or an **in-out** parameter. In each class of parameter, the responsibility for allocating and freeing non-by-value parameters is the following:

in parameter	Allocated and freed by the caller.
out parameter	Allocated by the callee; freed by the caller.
in-out parameter	Initially allocated by the caller, then freed and re-allocated by the callee if necessary. As with out parameters, the caller is responsible for freeing the final returned value.

In the latter two cases there is one piece of code that allocates the memory and a different piece of code that frees it. In order for this to be successful, the two pieces of code must of course have knowledge of which memory allocator is being used. Again, it is often the case that the two pieces of code are written by independent development organizations. To make this work, we require that the COM allocator be used.

Further, the treatment of out and in-out parameters in failure conditions needs special attention. If a function returns a status code which is a failure code, then in general the caller has no way to clean up the *out* or *in-out* parameters. This leads to a few additional rules:

out parameter	In error returns, out parameters must be *always* reliably set to a value which will be cleaned up without any action on the caller's part. Further, it is the case that all out pointer parameters (usually passed in a pointer-to-pointer parameter, but which can also be passed as a member of a caller-allocate callee-fill structure) *must* explicitly be set to NULL. The most straightforward way to ensure this is (in part) to set these values to NULL on function entry.[3] (On success returns, the semantics of the function of course determine the legal return values.)
in-out parameter	In error returns, all in-out parameters must either be left alone by the callee (and thus remaining at the value to which it was initialized by the caller; if the caller didn't initialize it, then it's an out parameter, not an in-out parameter) or be explicitly set as in the out parameter error return case.

The specific COM APIs and interfaces that apply to memory management are discussed further below.

Remember that these memory management conventions for COM applications apply only across public interfaces and APIs—there is no requirement at all that memory allocation strictly internal to a COM application need be done using these mechanisms.

1.3The COM Client/Server Model

Chapter 1 mentioned how COM supports a model of client/server interaction between a user of an object's services, the client, and the implementor of that object and its services, the server. To be more

[2] Any internally-used memory in COM and in-process objects can use any allocation scheme desired, but the COM memory allocator is a handy, efficient, and thread-safe allocator.

[3] This rule is stronger than it might seem to need to be in order to promote more robust application interoperability.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

precise, the client is *any* piece of code (not necessarily an application) that somehow obtains a pointer through which it can access the services of an object and then invokes those services when necessary. The server is some piece of code that implements the object and structures in such a way that the COM Library can match that implementation to a class identifier, or CLSID. The involvement of a class identifier is what differentiates a server from a more general object implementor.

The COM Library uses the CLSID to provide "implementation locator" services to clients. A client need only tell COM the CLSID it wants and the type of server—in-process, local, or remote—that it allows COM to load or launch. COM, in turn, locates the implementation of that class and establishes a connection between it and the client. This relationship between client, COM, and server is illustrated in Figure 2-2 on the next page.

Chapter 1 also introduced the idea of Location transparency, where clients and servers never need to know how far apart they actually are, that is, whether they are in the same process, different processes, or different machines.

This section now takes a closer look at the mechanisms in COM that make this transparency work as well as the responsibilities of client and server applications.

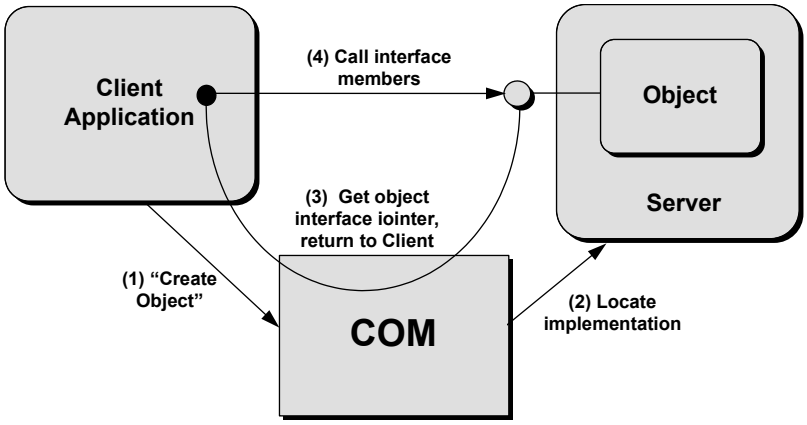


Figure 2-2: Clients locate and access objects through implementation locator services in COM. COM then connects the client to the object in a server. Compare this with Figure 1-2 in Chapter 1.

1.3.1COM Objects and Class Identifiers

A COM class is a particular implementation of certain interfaces; the implementation consists of machine code that is executed whenever you interact with an instance of the COM class. COM is designed to allow a class to be used by different applications, including applications written without knowledge of that particular class's existence. Therefore class code exists either in a dynamic linked library (DLL) or in another application (EXE). COM specifies a mechanism by which the class code can be used by many different applications.

A COM object is an object that is identified by a unique 128-bit CLSID that associates an object class with a particular DLL or EXE in the file system. A CLSID is a GUID itself (like an interface identifier), so no other class, no matter what vendor writes it, has a duplicate CLSID. Servers implementors generally obtain CLSIDs through the CoCreateGUID function in COM, or through a COM-enabled tool that internally calls this function.

The use of unique CLSIDs avoids the possibility of name collisions among classes because CLSIDs are in no way connected to the names used in the underlying implementation. So, for example, two different vendors can write classes which they call "StackClass," but each will have a unique CLSID and therefore avoid any possibility of a collision.

Further, no central authoritative and bureaucratic body is needed to allocate or assign CLSIDs. Thus, server implementors across the world can independently develop and deploy their software without fear of accidental collision with software written by others.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

On its host system, COM maintains a registration database (or "registry") of all the CLSIDs for the servers installed on the system, that is, a mapping between each CLSID and the location of the DLL or EXE that houses the server for that CLSID. COM consults this database whenever a client wants to create an instance of a COM class and use its services. That client, however, only needs to know the CLSID which keeps it independent of the specific location of the DLL or EXE on the particular machine.

If a requested CLSID is not found in the local registration database, various other administratively-controlled algorithms are available by which the implementation is attempted to be located on the network to which the local machine may be attached; these are explained in more detail below.

Given a CLSID, COM invokes a part of itself called the Service Control Manager (SCM[4]) which is the system element that locates the code for that CLSID. The code may exist as a DLL or EXE on the same machine or on another machine: the SCM isolates most of COM, as well as all applications, from the specific actions necessary to locate code. We'll return a discussion of the SCM in a moment after examining the roles of the client and server applications.

1.3.2COM Clients

Whatever application passes a CLSID to COM and asks for an instantiated object in return is a COM Client. Of course, since this client uses COM, it is also a COM application that must perform the required steps described above and in subsequent chapters.

Regardless of the type of server in use (in-process, local, or remote), a COM Client always asks COM to instantiate objects in exactly the same manner. The simplest method for creating one object is to call the COM function CoCreateInstance. This creates one object of the given CLSID and returns an interface pointer of whatever type the client requests. Alternately, the client can obtain an interface pointer to what is called the "class factory" object for a CLSID by calling CoGetClassObject. This class factory supports an interface called IClassFactory through which the client asks that factory to manufacture an object of its class. At that point the client has interface pointers for *two separate objects*, the class factory and an object of that class, that each have their own reference counts. It's an important distinction that is illustrated in Figure 2-3 and clarified further in Chapter 5.

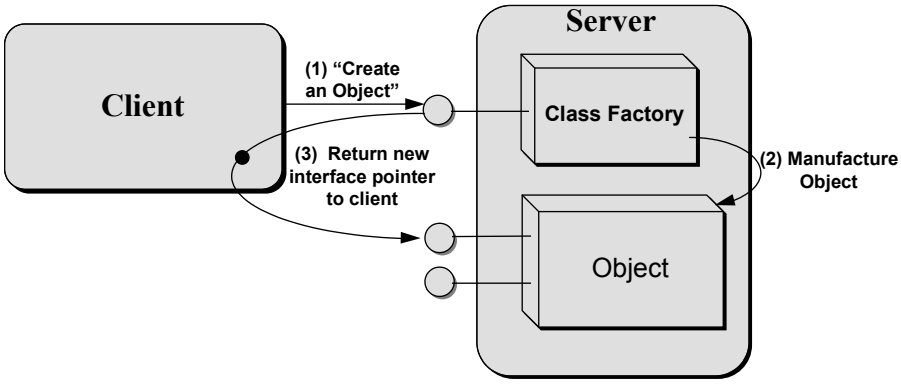


Figure 2-3: A COM Client creates objects through a class factory.

The CoCreateInstance function internally calls CoGetClassObject itself. It's just a more convenient function for clients that want to create one object.

The bottom line is that a COM Client, in addition to its responsibilities as a COM application, is responsible to use COM to obtain a class factory, ask that factory to create an object, initialize the object, and to call that object's (and the class factory's) Release function when the client is finished with it. These steps are the bulk of Chapter 5 which also explains some features of COM that allow clients to manage when servers are loaded and unloaded to optimize performance.

1.3.3COM Servers

There are two basic kinds of object servers:

[4] Colloquially, of course, pronounced "scum."

Copyright © 1995 Microsoft Corporation
All Rights Reserved

- **Dynamic Link Library (DLL) Based:** The server is implemented in a module that can be loaded into, and will execute within, a client's address space. (The term DLL is used in this specification to describe any shared library mechanism that is present on a given COM platform.)
- **EXE Based:** The server is implemented as a stand-alone executable module.

Since COM allows for distributed objects, it also allows for the two basic kinds of servers to implemented on a remote machine. To allow client applications to activate remote objects, COM defines the Service Control Manager (SCM) whose role is described below under "The COM Library."

As a client is responsible for using a class factory and for server management, a server is responsible for implementing the class factory, implementing the class of objects that the factory manufactures, exposing the class factory to COM, and providing for unloading the server under the right conditions. A diagram illustrating what exists inside a server module (EXE or DLL) is shown in Figure 2-4.

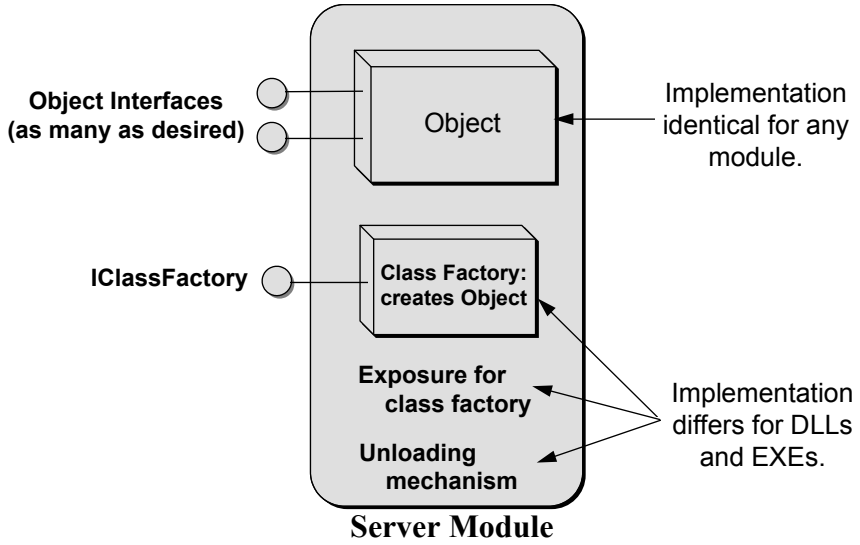


Figure 2-4: The general structure of a COM server.

How a server accomplishes these requirements depends on whether the server is implemented as a DLL or EXE, but is independent of whether the server is on the same machine as the client or on a remote machine. That is, remote servers are the same as local servers but have been registered to be visible to remote clients. Chapter 6 goes into all the necessary details about these implementations as well as how the server publishes its existence to COM in the registration database.

A special kind of server is called an "custom object handler" that works in conjunction with a local server to provide a partial in-process implementation of an object class.[5] Since in-process code is normally much faster to load, in-process calls are extremely fast, and certain resources can be shared only within a single process space, handlers can help improve performance of general object operations as well as the quality of operations such as printing. An object handler is architecturally similar to an in-process server but with more specialized semantics for its use. While the client can control the loading of handlers, it doesn't have to do any special work whatsoever to work with them. The existence of a handler changes nothing for clients.

1.3.4The COM Library and Service Control Manager

As described in Chapter 1, the COM Library itself is the implementation of the standard API functions defined in COM along with support for communicating between objects and clients. The COM Library is then the underlying "plumbing" that makes everything work transparently through RPC as shown in Figure 2-5 (this the same figure as Figure 1-8 in Chapter 1, repeated here for convenience). Whenever

[5] Strictly speaking, the "handler" is simply the representative of a remote object that resides in the client's process and which internally contains the remote connection. There is thus *always* a handler present when remoting is being done, though very often the handler is a trivial one which merely forwards all calls. In that sense, "handler" is synonymous with the terms "proxy object" or "object proxy." In practice the term "handler" tends to be used more when there is in fact a non-trivial handler, with "proxy" usually used when the handler is in fact trivial.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

COM determines that it has to establish communication between a client and a local or remote server, it creates "proxy" objects that act as in-process objects to the client. These proxies then talk to "stub" objects that are in the same process as the server and can call the server directly. The stubs pick up RPC calls from the proxies, turn them into function calls to the real object, then pass the return values back to the proxy via RPC which in turn returns them to the client.[6] The underlying remote procedure call mechanism is based on the standard DCE remote procedure call mechanism.

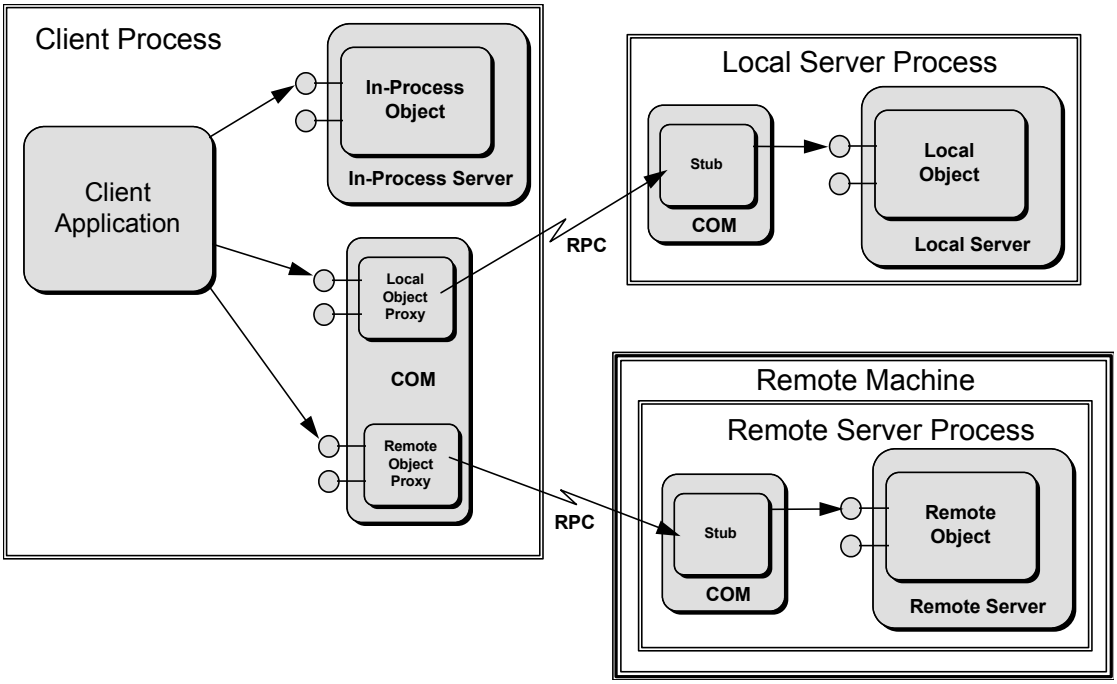


Figure 2-5: COM provides transparent access to local and remote servers through proxy and stub objects.

1.3.5Architecture for Distributed Objects

The COM architecture for object distribution is similar to the remoting architecture. When a client wants to connect to a server object, the name of the server is stored in the system registry. With distributed objects, the server can implemented as an in-process DLL, a local executable, or as executable or DLL running remotely. A component called the Service Control Manager (SCM) is responsible for locating the server and running it. The next section, "The Service Control Manager", explains the role of the SCM in greater depth and Chapter 15 contains the specification for it's interfaces.

Making a call to an interface method in a remote object involves the cooperation of several components. The interface proxy is a piece of interface-specific code that resides in the client's process space and prepares the interface parameters for transmittal. It packages, or marshals, them in such a way that they can be recreated and understood in the receiving process. The interface stub, also a piece of interface-specific code, resides in the server's process space and reverses the work of the proxy. The stub unpackages, or unmarshals, the sent parameters and forwards them on to the server. It also packages reply information to send back to the client.

The actual transmitting of the data across the network is handled by the RPC runtime library and the channel, part of the COM library. The channel works transparently with different channel types and supports both single and multi-threaded applications.

The flow of communication between the components involved in interface remoting is shown in Figure 2-6. On the client side of the process boundary, the client's method call goes through the proxy and then onto the channel. Note that the channel is part of the COM library. The channel sends the buffer

[6] Readers more familiar with RPC than with COM will recognize "client stub" and "server stub" rather than "proxy" and "stub" but the phrases are analogous.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

containing the marshaled parameters to the RPC runtime library who transmits it across the process boundary. The RPC runtime and the COM libraries exist on both sides of the process.

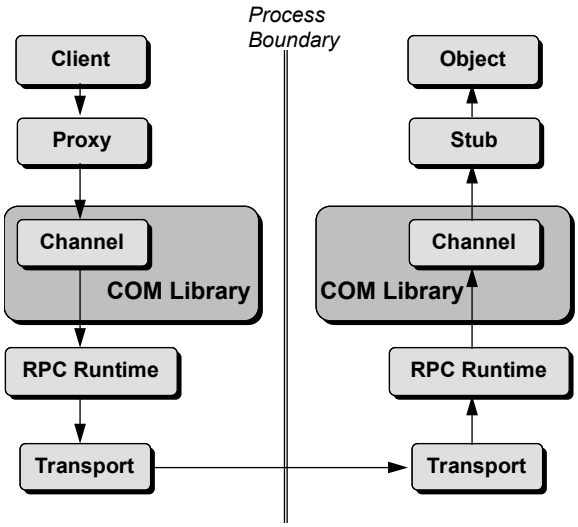


Figure 2-6. Components of COM's distributed architecture.

1.3.6 The Service Control Manager

The Service Control Manager ensures that when a client request is made, the appropriate server is connected and ready to receive the request. The SCM keeps a database of class information based on the system registry that the client caches locally through the COM library. This is the basis for COM's implementation locator services as shown in Figure 2-7.

When a client makes a request to create an object of a CLSID, the COM Library contacts the local SCM (the one on the same machine) and requests that the appropriate server be located or launched, and a class factory returned to the COM Library. After that, the COM Library, or the client, can ask the class factory to create an object.

The actions taken by the local SCM depend on the type of object server that is registered for the CLSID:

In-Process The SCM returns the file path of the DLL containing the object server implementation. The COM library then loads the DLL and asks it for its class factory interface pointer.

Local The SCM starts the local executable which registers a class factory on startup. That pointer is then available to COM.

Remote The local SCM contacts the SCM running on the appropriate remote machine and forwards the request to the remote SCM. The remote SCM launches the server which registers a class factory like the local server with COM on that remote machine. The remote SCM then maintains a connection to that class factory and returns an RPC connection to the local SCM which corresponds to that remote class factory. The local SCM then returns that connection to COM which creates a class factory proxy which will internally forward requests to the remote SCM via the RPC connection and thus on to the remote server.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

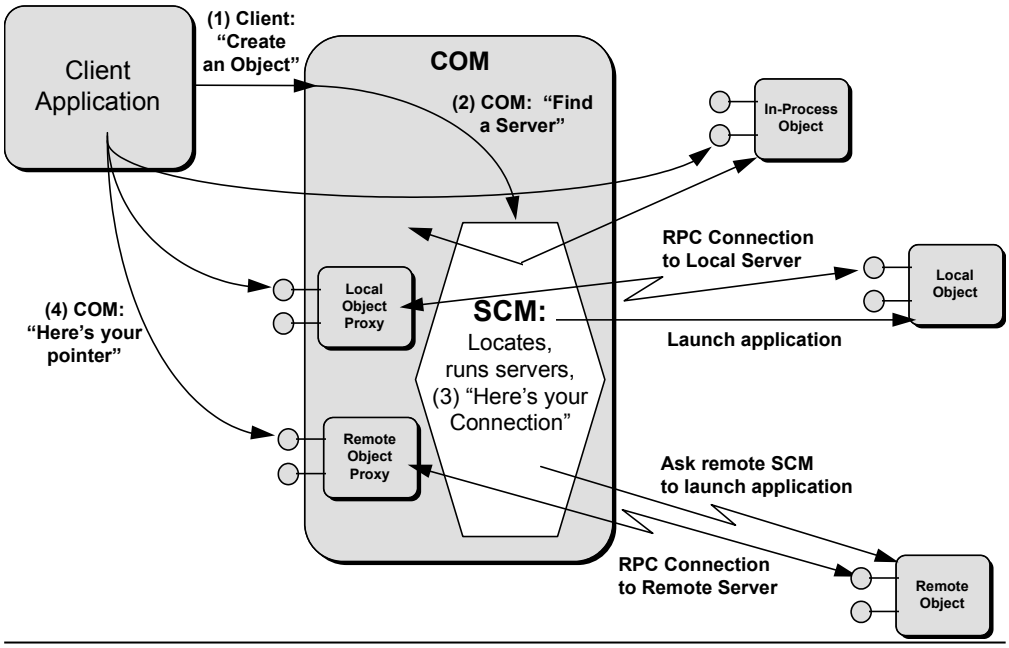


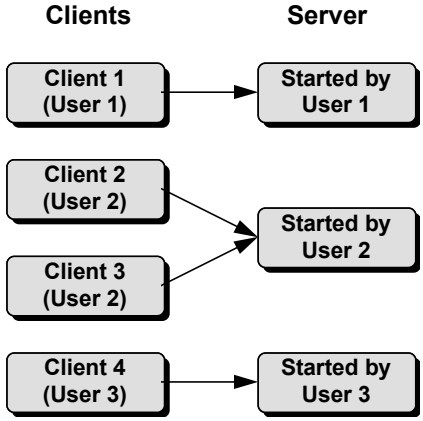
Figure 2-7: COM delegates responsibility of loading and launching servers to the SCM.Note that if the remote SCM determines that the remote server is actually an in-process server, it launches a "surrogate" server that then loads that in-process server. The surrogate does nothing more than pass all requests on through to the loaded DLL.

1.3.7Application Security

The technology in COM provides security for applications, regardless of whether they run remotely. There is a default level of security that is provided to non-security-aware applications such as existing OLE applications. Beyond the default, applications that are security-aware can control who is granted access to their services and the type of access that is granted.

Figure 2-8. A non-security-aware serverDefault security insures that system integrity is maintained. When multiple users require the services of a single non-security-aware server, a separate instance for each user is run. Each client/server connection remains independent from the others, preventing clients from accessing each others' data. All non-security-aware servers are run as the security principal who caused them to run. An example involving four clients that all require server "X" is illustrated in Figure 2-8. Since two of the clients are the same user (User2), one instance of server X can service both clients.

The technology used in COM for distribution implements this security system with the authentication services provided by RPC. These services are accessed by applications through the COM library when a call is made to CoInitialize. This security system imposes a restriction on where non-security-aware



All Rights Reserved

applications can run. Since the system cannot start a session on another machine without the proper credentials, all servers that run in the client security context normally run where their client is running. The AtBits attribute associated with that class controls where a server is run.

Security-aware servers are those applications that do not allow global access to their services. These servers may run either where the client is running, where their data is stored, or elsewhere depending on a rich set of activation rules. Rather than running as one of their clients; security-aware servers are themselves security principals. Security-aware servers may participate in two-way authentication whereby clients can ask for verification. Security-aware servers can use the services offered by the RPC security provider(s) or supply their own security implementation.

1.4Object Reusability

An important goal of any object model is that component authors can reuse and extend objects provided by others as pieces of their own component implementations. Implementation inheritance is one way this can be achieved: to reuse code in the process of building a new object, you inherit implementation from it and override methods in the tradition of C++ and other languages. However, as a result of many years experience, many people believe traditional language-style implementation inheritance technology as the basis for object reuse is simply not robust enough for large, evolving systems composed of software components. (See page Error: Reference source not found for more information.) For this reason COM introduces other reusability mechanisms.

1.4.1COM 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 does not need to know anything, 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*.

To achieve black-box reusability, COM supports two mechanisms through which one object 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.”

1. **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’s services to implement itself. It is not necessary that the outer and inner objects support the same interfaces; in fact, the outer object may use an inner object’s interface to help implement parts of a different interface on the outer object especially when the complexity of the interfaces differs greatly.

2. **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 2-9 and 2-10. The important part to 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.

Containment is simple to implement for an outer object: during its creation, the outer object creates whatever inner objects it needs to use as any other client would. This is nothing new—the process is like a C++ object that itself contains a C++ string object that it uses to perform certain string functions even if the outer object is not considered a “string” object in its own right.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

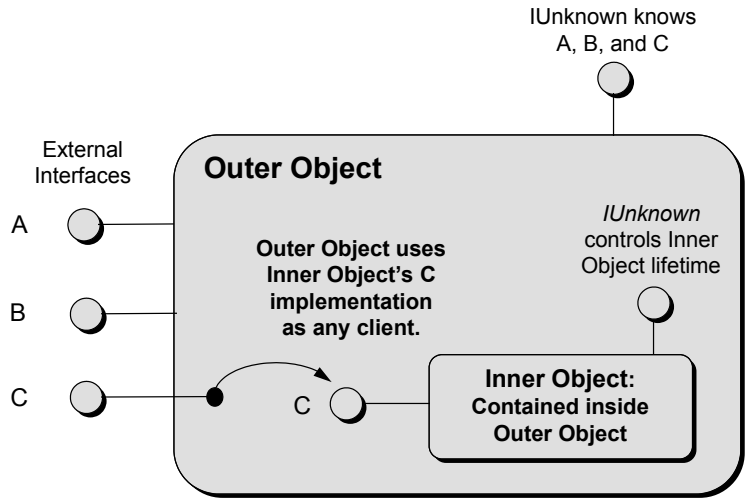


Figure 2-9: Containment of an inner object and delegation to its interfaces.

Aggregation is almost as simple to implement, the primary difference being the implementation of the three IUnknown functions: QueryInterface, AddRef, and Release. The catch is that from the client's perspective, any IUnknown function on the outer object must affect the outer object. That is, AddRef and Release affect the outer object and QueryInterface exposes all the interfaces available on the outer object. However, if the outer object simply exposes an inner object's interface as it's own, that inner object's IUnknown members called through that interface will behave differently than those IUnknown members on the outer object's interfaces, a sheer violation of the rules and properties governing IUnknown.

The solution is for the outer object to somehow pass the inner object some IUnknown pointer to which the inner object can re-route (that is, delegate) IUnknown calls in its own interfaces, and yet there must be a method through which the outer object can access the inner object's IUnknown functions that only affect the inner object. COM provides specific support for this solution as described in Chapter 6.

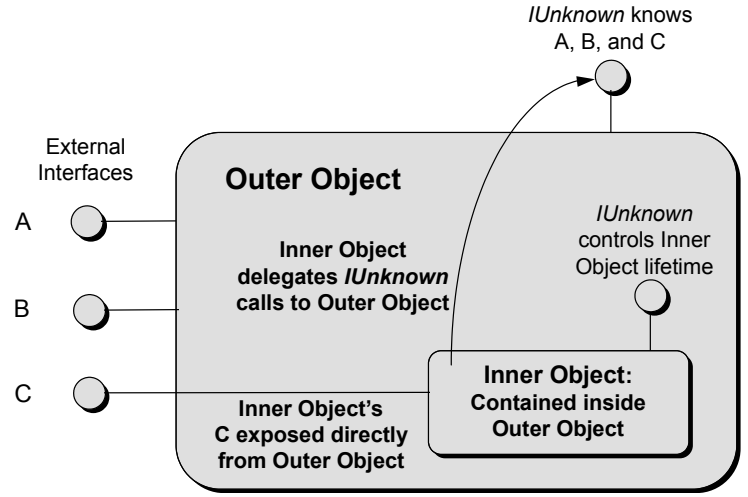


Figure 2-10: Aggregation of an inner object where the outer object exposes one or more of the inner object's interfaces as it's own.

1.5Connectable Objects and Events

In the preceding discussions of interfaces it was implied that, from the object's perspective, the interfaces were "incoming". "Incoming," in the context of a client-object relationship, implies that the object "listens" to what the client has to say. In other words, incoming interfaces and their member functions receive input from the outside. COM also defines mechanisms where objects can support "outgoing" interfaces. Outgoing interfaces allow objects to have two-way conversations, so to speak, with clients.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

When an object supports one or more outgoing interfaces, it is said to be *connectable.* One of the most obvious uses for outgoing interfaces is for event notification. This section describes Connectable Objects.[7]

A connectable object (also called a *source*) can have as many outgoing interfaces as it likes. Each interface is composed of distinct member functions, with each function representing a single *event, notification,* or *request.* Events and notifications are equivalent concepts (and interchangeable terms), as they are both used to tell the client that something interesting happened in the object. Events and notifications differ from a request in that the object expects response from the client. A request, on the other hand, is how an object asks the client a question and expects a response.

In all of these cases, there must be some client that listens to what the object has to say and uses that information wisely. It is the client, therefore, that actually implements these interfaces on objects called *sinks*. From the sink's perspective, the interfaces are incoming, meaning that the sink listens through them. A connectable object plays the role of a client as far as the sink is concerned; thus, the sink is what the object's client uses to listen to that object.

An object doesn't necessarily have a one-to-one relationship with a sink. In fact, a single instance of an object usually supports any number of connections to sinks in any number of separate clients. This is called *multicasting.*[8] In addition, any sink can be connected to any number of objects.

Chapter 11 covers the Connectable Object interfaces (IConnectionPoint and IConnectionPointContainer) in complete detail.

1.6Persistent Storage

As mentioned in Chapter 1, the enhanced COM services define a number of storage-related interfaces, collectively called Persistent Storage or Structured Storage. By definition of the term *interface*, these interfaces carry no implementation. They describe a way to create a "file system within a file," and they provide some extremely powerful features for applications including incremental access, transactioning, and a sharable medium that can be used for data exchange or for storing the persistent data of objects that know how to read and write such data themselves. The following sections deal with the structure of storage and the other features.

1.6.1A File System Within A File

Years ago, before there were "disk operating systems," applications had to write persistent data directly to a disk drive (or drum) by sending commands directly to the hardware disk controller. Those applications were responsible for managing the absolute location of the data on the disk, making sure that it was not overwriting data that was already there. This was not too much of a problem seeing as how most disks were under complete control of a single application that took over the entire computer.

The advent of computer systems that could run more than one application brought about problems where all the applications had to make sure they did not write over each other's data on the disk. It therefore became beneficial that each adopted a standard of marking the disk sectors that were used and which ones were free. In time, these standards became the "disk operating system" which provided a "file system." Now, instead of dealing directly with absolute disk sectors and so forth, applications simply told the file system to write blocks of data to the disk. Furthermore, the file system allowed applications to create a hierarchy of information using directories which could contain not only files but other sub-directories which in turn contained more files, more sub-directories, etc.

The file system provided a single level of indirection between applications and the disk, and the result was that every application saw a file as a single contiguous stream of bytes on the disk. Underneath, however, the file system was storing the file in dis-contiguous sectors according to some algorithm that optimized read and write time for each file. The indirection provided from the file system freed applications from having to care about the absolute position of data on a storage device.

Today, virtually all system APIs for file input and output provide applications with some way to write information into a flat file that applications see as a single stream of bytes that can grow as large as

[7] OLE Controls use the Connectable Objects mechanisms extensively.

[8] Note that this usage of the term *multicasting* may differ from what some readers are accustomed to. In some systems *multicasting* is used to describe a connection-less broadcast. Connectable objects are obviously connection oriented.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

necessary until the disk is full. For a long time these APIs have been sufficient for applications to store their persistent information. Applications have made some incredible innovations in how they deal with a single stream of information to provide features like incremental "fast" saves.

However, a major feature of COM is interoperability, the basis for integration between applications. *This integration brings with it the need to have multiple applications write information to the same file on the underlying file system.* This is exactly the same problem that the computer industry faced years ago when multiple applications began to share the same disk drive. The solution then was to create a file system to provide a level of indirection between an application "file" and the underlying disk sectors.

Thus, the solution for the integration problem today is another level of indirection: a file system *within* a file. Instead of requiring that a large contiguous sequence of bytes on the disk be manipulated through a single file handle with a single seek pointer, COM defines how to treat a single file system entity as a structured collection of two types of objects—storages and streams—that act like directories and files, respectively.

1.6.2Storage and Stream Objects

Within COM's Persistent Storage definition there are two types of storage elements: storage objects and stream objects. These are objects generally implemented by the COM library itself; applications rarely, if ever, need to implement these storage elements themselves.[9] These objects, like all others in COM, implement interfaces: IStream for stream objects, IStorage for storage objects as detailed in Chapter 8.

A stream object is the conceptual equivalent of a single disk file as we understand disk files today. Streams are the basic file-system component in which data lives, and each stream in itself has access rights and a single seek pointer. Through its *IStream* interface stream can be told to read, write, seek, and perform a few other operations on its underlying data. Streams are named by using a text string and can contain any internal structure you desire because they are simply a flat stream of bytes. In addition, the functions in the *IStream* interface map nearly one-to-one with standard file-handle based functions such as those in the ANSI C run-time library.

A storage object is the conceptual equivalent of a directory. Each storage, like a directory, can contain any number of sub-storages (sub-directories) and any number of streams (files). Furthermore, each storage has its own access rights. The IStorage interface describes the capabilities of a storage object such as enumerate elements (dir), move, copy, rename, create, destroy, and so forth. A storage object itself cannot store application-defined data except that it implicitly stores the names of the elements (storages and streams) contained within it.

Storage and stream objects, when implemented by COM as a standard on a system, are sharable between processes. This is a key feature that enables objects running in-process or out-of-process to have equal incremental access to their on-disk storage. Since COM is loaded into each process separately, it must use some operating-system supported shared memory mechanisms to communicate between processes about opened elements and their access modes.

1.6.3Application Design with Structured Storage

COM's structured storage built out of storage and stream objects makes it much easier to design applications that by their nature produce structured information. For example, consider a "diary" program that allows a user to make entries for any day of any month of any year. Entries are made in the form of some kind of object that itself manages some information. Users wanting to write some text into the diary would store a text object; if they wanted to save a scan of a newspaper clip they could use a bitmap objects, and so forth.

[9] This specification recommends that the COM implementation on a given platform (Windows, Macintosh, etc.) includes a standard storage implementation for use by all applications.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Without a powerful means to structure information of this kind, the diary application might be forced to manage some hideous file structure with an overabundance of file position cross-reference pointers as shown in Figure 2-11.

There are many problems in trying to put structured information into a flat file. First, there is the sheer tedium of managing all the cross-reference pointers in all the different structures of the file. Whenever a piece of information grows or moves in the file, every cross-reference offset referring to that information must be updated as well. Therefore even a small change in the size of one of the text objects or an addition of a day or month might precipitate changes throughout the rest of the file to update seek offsets. While not only tedious to manage, the application will have to spend enormous amounts of time moving information around in the file to make space for data that expands. That, or the application can move the newly enlarged data to the end of the file and patch a few seek offsets, but that introduces the whole problem of garbage collection, that is, managing the free space created in the middle of the file to minimize waste as well as overall file size.

The problems are compounded even further with objects that are capable of reading and writing their own information to storage. In the example here, the diary application would prefer to give each objects in it —text, bitmap, drawing, table, etc.—its own piece of the file in which the object can write whatever the it wants, however much it wants. The only practical way to do this with a single flat file is for the diary application to ask each object for a memory copy of what the object would like to store, and then the diary would write that information into a place in its own file. This is really the only way in which the diary could manage the location of all the information. Now while this works reasonably well for small data, consider an object that wants to store a 10MB bitmap scan of a true-color photograph—exchanging that much data through memory is horribly inefficient. Furthermore, if the end user wants to later make changes to that bitmap, the diary would have to load the bitmap *in entirety* from its file and pass it back to the object. This is again extraordinarily inefficient.[10]

COM's Persistent Storage technology solves these problems through the extra level of indirection of a

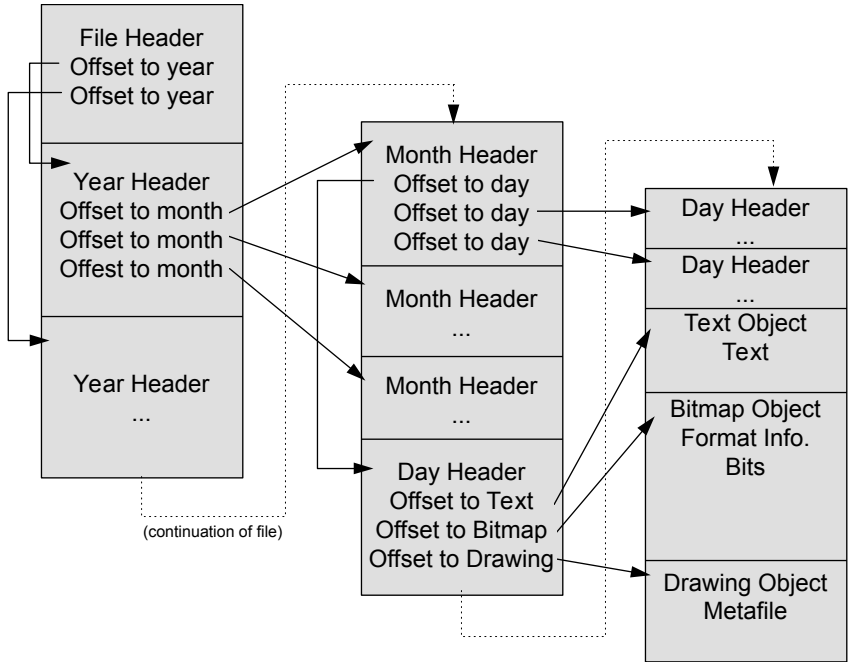


Figure 2-11: A flat-file structure for a diary application. This sort of structure is difficult to manage.

file system within a file. With COM, the diary application can create a structured hierarchy where the root file itself has sub-storages for each year in the diary. Each year sub-storage has a sub-storage for each month, and each month has a sub-storage for each day. Each day then would have yet another sub-

[10] This mechanism, in fact, was employed by compound documents in Microsoft's OLE version 1.0. The problems describe here were some of the major limitations of OLE 1.0 which provided much of the impetus for COM's Persistent Storage technology.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

storage or perhaps just a stream for each piece of information that the user stores in that day.[11] This configuration is illustrated in Figure 2-12.

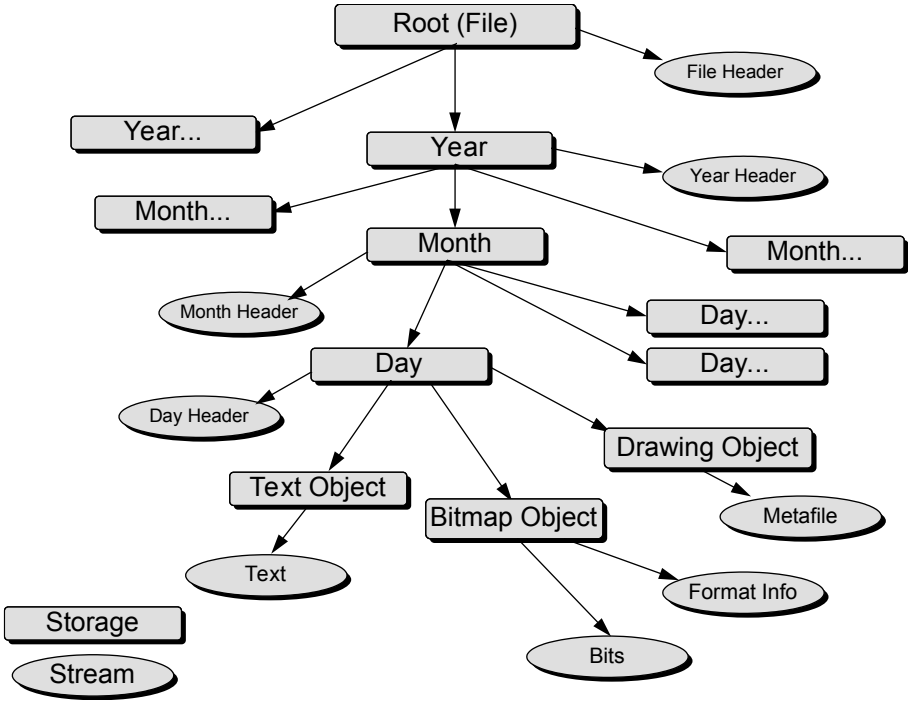


Figure 2-12: A structured storage scheme for a diary application. Every object that has some content is given its own storage or stream element for its own exclusive use.

This structure solves the problem of expanding information in one of the objects: the object itself expands the streams in its control and the COM implementation of storage figures out where to store all the information in the stream. The diary application doesn't have to lift a finger. Furthermore, the COM implementation automatically manages unused space in the entire file, again, relieving the diary application of a great burden.

In this sort of storage scheme, the objects that manage the content in the diary always have direct *incremental* access to their piece of storage. That is, when the object needs to store its data, it writes it *directly* into the diary file without having to involve the diary application itself. The object can, if it wants to, write *incremental changes* to that storage, thus leading to much better performance than the flat file scheme could possibly provide. If the end user wanted to make changes to that information later on, the object can then incrementally read as little information as necessary instead of requiring the diary to read all the information into memory first. Incremental access, a feature that has traditionally been very hard to implement in applications, is now the *default mode of operation*. All of this leads to much better performance.

1.6.4Naming Elements

Every storage and stream object in a structured file has a specific character name to identify it. These names are used to tell *IStorage* functions what element in that storage to open, destroy, move, copy, rename, etc. Depending on which component, client or object, actually defines and stores these names, different conventions and restrictions apply.

Names of root storage objects are in fact names of files in the underlying file system. Thus, they obey the conventions and restrictions that it imposes. Strings passed to storage-related functions which name files are passed on un-interpreted and unchanged to the file system.

Names of elements contained within storage objects are managed by the implementation of the particular storage object in question. All implementations of storage objects must at the least support element

[11] The application would only create year, month, and day substorages for those days that had information in them, that is, the diary application would create sparse storage for efficiency.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

names that are 32 characters in length; some implementations may if they wish choose to support longer names. Names are stored case-preserving, but are compared case-insensitive.[12] As a result, applications which define element names must choose names which will work in either situation.

The names of elements inside an storage object must conform to certain conventions:

1. The two specific names "." and ".." are reserved for future use.
2. Element names cannot contain any of the four characters "\", "/", ":", or "!".

In addition, the name space in a storage element is partitioned in to different areas of ownership. Different pieces of code have the right to create elements in each area of the name space.

- The set of element names beginning with characters other than '\0x01' through '\0x1F' (that is, decimal 1 through decimal 31) are for use by the object whose data is stored in the *IStorage*. Conversely, the object must *not* use element names beginning with these characters.
- Element names beginning with a '\0x01' and '\0x02' are for the exclusive use of COM and other system code built on it such as OLE Documents.
- Element names beginning with a '\0x03' are for the exclusive use of the client which is managing the object. The client can use this space as a place to persistently store any information it wishes to associate with the object along with the rest of the storage for that object.
- Element names beginning with a '\0x04' are for the exclusive use of the COM structured storage implementation itself. They will be useful, for example, should that implementation support other interfaces in addition to IStorage, and these interface need persistent state.
- Element names beginning with '\0x05' and '\0x06' are for the exclusive use of COM and other system code built on it such as OLE Documents.
- All other names beginning with '\0x07' through '\0x1F' are reserved for future definition and use by the system.

In general, an element's name is not considered useful to an end-user. Therefore, if a client wants to store specific user-readable names of objects, it usually uses some other mechanism. For example, the client may write its own stream under one of its own storage elements that has the names of all the other objects within that same storage element. Another method would be for the client to store a stream named "\0x03Name" in each object's storage that would contain that object's name. Since the stream name itself begins with '\0x03' the client owns that stream even through the objects controls much of the rest of that storage element.

1.6.5Direct Access vs. Transacted Access

Storage and stream elements support two fundamentally different modes of access: direct mode and transacted mode. Changes made while in direct mode are immediately and permanently made to the affected storage object. In transacted mode, changes are buffered so that they may be saved ("committed") or reverted when modifications are complete.

If an outermost level IStorage is used in transacted mode, then when it commits, a robust two-phase commit operation is used to publish those changes to the underlying file on the file system. That is, great pains are taken are taken so as not to loose the user's data should an untimely crash occurs.

The need for transacted mode is best explained by an illustrative scenario. Imagine that a user has created a spreadsheet which contains a sound clip object, and that the sound clip is an object that uses the new persistent storage facilities provided in COM. Suppose the user opens the spreadsheet, opens the sound clip, makes some editing changes, then closes the sound clip at which point the changes are updated in the spreadsheet storage set aside for the sound clip. Now, at this instant, the user has a choice: save the spreadsheet or close the spreadsheet *without* saving. Either way, the next time the user opens the spreadsheet, the sound clip had better be in the appropriate state. This implies that at the instant before the save vs. close decision was made, both the old and the new versions of the sound clip had to exist. Further, since large objects are precisely the ones that are expensive in time and space to copy, the new version should exist as a set of *differences* from the old.

[12] Case sensitivity is a locale-sensitive operation: some characters compare case-insenstive-equal in some locales and -not-equal in others. In an IStorage implementation, the case-insenstive comparision is done with respect to the current locale in which the system is presently running. This has implications on the use of IStorage names for those who wish to create globally portable documents.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

The central issue is whose responsibility it is to keep track of the two versions. The client (the spreadsheet in this example) had the old version to begin with, so the question really boils down to how and when does the object (sound clip) communicate the new version to the spreadsheet. Applications today are in general already designed to keep edits separate from the persistent copy of an object until such time as the user does a save or update. Update time is thus the earliest time at which the transfer should occur. The latest is immediately before the client saves itself. The most appropriate time seems to be one of these two extremes; no intermediate time has any discernible advantage.

COM specifies that this communication happens at the earlier time. When asked to update edits back to the client, an object using the new persistence support will write any changes to its storage) exactly as if it were doing a save to its own storage completely outside the client. It is the responsibility of the client to keep these changes separate from the old version until *it* does a save (commit) or close (revert). Transacted mode on *IStorage* makes dealing with this requirement easy and efficient.

The transaction on each storage is nested in the transaction of its parent storage. Think of the act of committing a transaction on an IStorage instance as "publishing changes one more level outwards." Inner objects publish changes to the transaction of the next object outwards; outermost objects publish changes permanently into the file system.

Let's examine for a moment the implications of using instead the second option, where the object keeps all editing changes to itself until it is known that the user wants to commit the client (save the file). This may happen many minutes after the contained object was edited. COM must therefore allow for the possibility that in the interim time period the user closed the server used to edit the object, since such servers may consume significant system resources. To implement this second option, the server must presumably keep the changes to the old version around in a set of temporary files (remember, these are potentially *big* objects). At the client's commit time, every server would have to be restarted and asked to incorporate any changes back onto its persistent storage. This could be *very* time consuming, and could significantly slow the save operation. It would also cause reliability concern in the user's mind: what if for some reason (such as memory resources) a server cannot be restarted? Further, even when the client is closed *without* saving, servers have to be awakened to clean up their temporary files. Finally, if a object is edited a second time before the client is committed, in this option its the client can only provide the *old, original* storage, not the storage that has the first edits. Thus, the server would have to recognize on startup that some edits to this object were lying around in the system. This is an awkward burden to place on servers: it amounts to requiring that they *all* support the ability to do incremental auto-save with automatic recovery from crashes. In short, this approach would significantly and unacceptably complicate the responsibilities of the object implementors.

To that end, it makes the most sense that the standard COM implementation of the storage system support transactioning through IStorage and possibly IStream.

1.6.6 Browsing Elements

By its nature, COM's structured storage separates applications from the exact layout of information within a given file. Every element of information in that file is access using functions and interfaces implemented by COM. Because this implementation is central, a file generated by some application using this structure can be browsed by some other piece of code, such as a system shell. In other words, any piece of code in the system can use COM to browse the entire hierarchy of elements within any structured file simply by navigating with the IStorage interface functions which provide directory-like services. If that piece of code also knows the format and the meaning of a specific stream that has a certain name, it could also *open* that stream and make use of the information in it, *without having to run the application that wrote the file.*

This is a powerful enabling technology for operating system shells that want to provide rich query tools to help end users look for information on their machine or even on a network. To make it really happen requires standards for certain stream names and the format of those streams such that the system shell can open the stream and execute queries against that information. For example, consider what is possible if all applications created a stream called "Summary Information" underneath the root storage element of the file. In this stream the application would write information such as the author of the document, the create/modify/last saved time-stamps, title, subject, keywords, comments, a thumbnail sketch of the first page, etc. Using this information the system shell could find any documents that a certain user write

Copyright © 1995 Microsoft Corporation
All Rights Reserved

before a certain date or those that contained subject matter matched against a few keywords. Once those documents are found, the shell can then extract the title of the document along with the thumbnail sketch and give the user a very engaging display of the search results.

This all being said, in the general the actual utility of this capability is perhaps significantly less than what one might first imagine. Suppose, for example, that I have a structured storage that contains some word processing document whose semantics and persistent representation I am unaware of, but which contains some number of contained objects, perhaps the figures in the document, that I can identify by their being stored and tagged in contained sub-storages. One might naively think that it would be reasonable to be able to walk in and browse the figures from some system-provided generic browsing utility. This would indeed work from a technical point of view; however, it is unlikely to be useable from a user interface perspective. The document may contain hundreds of figures, for example, that the user created and thinks about not with a name, not with a number, but only in the relationship of a particular figure to the rest of the document's information. *With what user interface could one reasonably present this list of objects to the user other than as some add-hoc and arbitrarily-ordered sequence?* There is, for example, no name associated with each object that one could use to leverage a file-system directory-browsing user interface design. In general, the *content* of a document can only be reasonably be presented to a human being using a tool that understands the semantics of the document content, and thus can show all of the information therein in its appropriate context.

1.6.7Persistent Objects

Because COM allows an object to read and write itself to storage, there must be a way through which the client tells objects to do so. The way is, of course, additional interfaces that form a storage contract between the client and objects. When a client wants to tell and object to deal with storage, it queries the object for one of the persistence-related interfaces, as suits the context. The interfaces that objects can implement, in any combination, are described below:

IPersistStorage	Object can read and write its persistent state to a storage object. The client provides the object with an IStorage pointer through this interface. This is the only IPersist* interface that includes semantics for incremental access.
IPersistStream	Object can read and write its persistent state to a stream object. The client provides the object with an IStream pointer through this interface.
IPersistFile	Object can read and write its persistent state to a file on the underlying system directly. This interface does not involve IStorage or IStream unless the underlying file is itself access through these interfaces, but the IPersistFile itself has no semantics relating to such structures. The client simply provides the object with a filename and orders to save or load; the object does whatever is necessary to fulfill the request.

These interfaces and the rules governing them are described in Chapter 12.

1.7Persistent, Intelligent Names: Monikers

To set the context for why "Persistent, Intelligent Names" are an important technology in COM, think for a moment about a standard, mundane file name. That file name refers to some collection of data that happens to be stored on disk somewhere. The file name describes the somewhere. In that sense, the file name is really a name for a particular "object" of sorts where the object is defined by the data in the file.

The limitation is that a file name by itself is unintelligent; all the intelligence about what that filename means and how it gets used, as well as how it is stored persistently if necessary, is contained in whatever application is the client of that file name. The file name is nothing more than some piece of data in that client. This means that the client must have specific code to handle file names. This normally isn't seen as much of a problem—most applications can deal with files and have been doing so for a long time.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

Now introduce some sort of name that describes a query in a database. The introduce others that describe a file and a specific range of data within that file, such as a range of spreadsheet cells or a paragraph is a document. Introduce yet more than identify a piece of code on the system somewhere that can execute some interesting operation. In a world where clients have to know what a name means in order to use it, those clients end up having to write specific code for each type of name causing that application to grow monolithically in size and complexity. This is one of the problems that COM was created to solve.

In COM, therefore, the intelligence of how to work with a particular name is encapsulated inside the name itself, where the name becomes an object that implements name-related interfaces. These objects are called *monikers*.[13] A moniker implementation provides an abstraction to some underlying connection (or "binding") mechanism. Each different moniker class (with a different CLSID) has its own semantics as to what sort of object or operation it can refer to, which is *entirely* up to the moniker itself. A section below describes some typical types of monikers. While a moniker class itself defines the operations necessary to locate some general type of object or perform some general type of action, each individual moniker *object* (each instantiation) maintains its own name data that identifies some other *particular* object or operation. The moniker class defines the functionality; a moniker object maintains the parameters.

With monikers, clients always work with names through an interface, rather than directly manipulating the strings (or whatever) themselves. This means that whenever a client wishes to perform any operation with a name, it calls some code to do it instead of doing the work itself. This level of indirection means that the moniker can transparently provide a whole host of services, and that the client can seamlessly interoperate over time with various different moniker implementations which implement these services in different ways.

1.7.1Moniker Objects

A moniker is simply an object that supports the IMoniker interface. IMoniker interface includes the IPersist-Stream interface;[14] thus, monikers can be saved to and loaded from streams. The persistent form of a moniker includes the data comprising its name and the CLSID of its implementation which is used during the loading process. This allows new kinds of monikers to be created transparently to clients.

The most basic operation in the IMoniker interface is that of *binding* to the object to which it points. The binding function in IMoniker takes as a parameter the interface identifier by which the client wishes to talk to the bound object, runs whatever algorithm is necessary in order to locate the object, then returns a pointer of that interface type to the client. The client can also ask to bind to the object's *storage* (for example, the IStorage containing the object) if desired, instead of to the running object through a slightly different IMoniker function. As binding may be an expensive and time-consuming process, a client can control how long it is willing to wait for the binding to complete. Binding also takes place inside a specific "bind context" that is given to the moniker. Such a context enables the binding process overall to be more efficient by avoiding repeated connections to the same object.

A moniker also supports an operation called "reduction" through which it re-writes itself into another equivalent moniker that will bind to the same object, but does so in a more efficient way. This capability is useful to enable the construction of user-defined macros or aliases as new kinds of moniker classes (such that when reduced, the moniker to which the macro evaluates is returned) and to enable construction of a kind of moniker which tracks data as it moves about (such that when reduced, the new moniker contains a reference to the new location). Chapter 9 will expand on the reduction concept.

Each moniker class can store arbitrary data its persistent representation, and can run arbitrary code at binding time. The client therefore only knows each moniker by the presence of a persistent representation and whatever label the client wishes to assign to each moniker. For example, a spreadsheet as a client may keep, from the user's perspective, a list of "links" to other spreadsheets where, in fact, each link was an arbitrary label for a moniker (regardless of whether the moniker is loaded or persistently on disk at the moment) where the moniker manages the real identity of the linked data. When the spreadsheet wants to resolve a link for the user, it only has to ask the moniker to bind to the object. After the binding is

[13] The word "moniker" is fairly obscure synonym for "nickname."

[14] One of the few instances of inheritance from one major interface to another, which the *IMoniker* designer later decided was actually less preferable to having a moniker implement *IMoniker* and *IPersistStream* separately. See the first footnote in Chapter 9.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

complete, the spreadsheet then has an interface pointer for the linked object and can talk to it directly—the moniker falls out of the picture as its job is complete.

The label assigned to a moniker by a client does not have to be arbitrary. Monikers support the ability to produce a "display name" for whatever object they represent that is suitable to show to an end user. A moniker that maintains a file name (such that it can find an application to load that file) would probably just use the file name directly as the display name. Other monikers for things such as a query may want to provide a display name that is a little more readable than some query languages.

1.7.2 Types of Monikers

As some of the examples above has hinted, monikers can have many types, or classes, depending on the information they contain and the type of objects they can refer to. A moniker class is really defined by the information it persistently maintains and the binding operation is uses on that information.

COM itself, however, only specifies one standard moniker called the *generic composite moniker*. The composite moniker is special in two ways. First, its persistent data is *completely* composed of the persistent data of other monikers, that is, a composite moniker is a collection of other monikers. Second, binding a composite moniker simply tells the composite to bind each moniker it contains in sequence. Since the composite's behavior and persistent state is defined by other monikers, it is a standard type of moniker that works identically on any host system; the composite is *generic* because it has no knowledge of its pieces except that they are monikers. Chapter 9 described the generic composite in more detail.

So what other types of monikers can go in a composite? Virtually any other type (including other composite monikers!). However, other types of monikers are not so generic and have more dependency on the underlying operating system or the scenarios in which such a moniker is used.

For example, Microsoft's OLE defines four other specific monikers—file, item, anti, pointer—that it uses specifically to help implement "linked objects" in its compound document technology. A file moniker, for example, maintains a file name as its persistent data and its binding process is one of locating an application that can load that file, launching the application, and retrieving from it an IPersistFile interface through which the file moniker can ask the application to load the file. Item monikers are used to describe smaller portions of a file that might have been loaded with a file moniker, such as a specific sheet of a three-dimensional spreadsheet or a range of cells in that sheet. To "link" to a specific cell range in a specific sheet of a specific file, the single moniker used to describe the link is a generic composite that is composed with a file moniker and two item monikers as illustrated in Figure 2-13. Each moniker in the composite is one step in the path to the final source of the link.

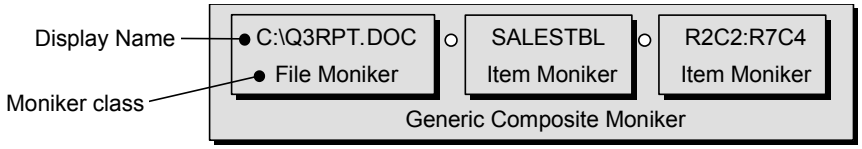


Figure 2-13: A composite moniker that is composed with a file moniker and two item monikers to describe the source of a link which is a cell range in a specific sheet of a spreadsheet file.

More complete descriptions of the file, item, anti, and pointer monikers from OLE are provided in Chapter 9 as examples of how monikers can be used. But monikers can represent virtually any type of information and operation, and are not limited to this basic set of OLE defined monikers.

1.7.3 Connections and Reconnections

How does a client come by a moniker in the first place? In other words, how does a client establish a connection to some object and obtain a moniker that describes that connection? The answer depends on the scenario involved but is generally one of two ways. First, the source of the object may have created a moniker and made it available for consumption through a data transfer mechanism such (in the workstation case) as a clipboard or perhaps a drag & drop operation. Second, the client may have enough knowledge about a particular moniker class that it can synthesize a moniker for some object using other known information such that the client can forget about that specific information itself and thereafter deal only with monikers. So regardless of how a client obtains a moniker, it can simply ask the moniker to bind to establish a connection to the object referred to by the moniker.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

Binding a moniker does not always mean that the moniker must run the object itself. The object might already be running within some appropriate scope (such as the current desktop) by the time the client wants to bind the moniker to it. Therefore the moniker need only connect to that running object.

COM supports this scenario through two mechanisms. The first is the *Running Object Table* in which objects register themselves and their monikers when they become running. This table is available to all monikers as they attempt to bind—if a moniker sees that a matching moniker in the table, it can quickly connect to the already running object.

1.8Uniform Data Transfer

Just as COM provides interfaces for dealing with storage and object naming, it also provides interfaces for exchanging data between applications. So built on top of both COM and the Persistent Storage technology is Uniform Data Transfer, which provides the functionality to represent all data transfers through a single implementation of a *data object*. Data objects implement an interface called IDataObject which encompasses the standard operations of get/set data and query/enumerate formats as well as functions through which a client of a data object can establish a notification loop to detect data changes in the object. In addition, this technology enables use of richer descriptions of data formats and the use of virtually any storage medium as the transfer medium.

1.8.1Isolation of Transfer Protocols

The "Uniform" in the name of this technology arose from the fact that the IDataObject interface separates all the common exchange operations from what is called a *transfer protocol*. Existing protocols include facilities such as a "clipboard" or a "drag & drop" feature as well as compound documents as implemented in OLE. With Uniform Data Transfer, all protocols are concerned only with exchanging a pointer to an IDataObject interface. The source of the data—the server—need only implement one data object which is usable in any exchange protocol and that's it. The consumer—the client—need only implement one piece of code to request data from a data object once it receives an IDataObject pointer from any protocol. Once the pointer exchange has occurred, both sides deal with data exchange in a uniform fashion, through IDataObject.

This uniformity not only reduces the code necessary to source or consume data, but also greatly simplifies the code needed to work with the protocol itself. Before COM was first implemented in OLE 2, each transfer protocol available on Microsoft Windows had its own set of functions that tightly bound the protocol to the act of requesting data, and so programmers had to implement specific code to handle each different protocol and exchange procedure. Now that the exchange functionality is separated from the protocol, dealing with each protocol requires only a minimum amount of code which is absolutely necessary for the semantics of that protocol.

While of course extremely useful in the context of OLE Documents, Uniform Data Transfer is a generic service with applications far beyond OLE Documents.

1.8.2Data Formats and Transfer Mediums

Before Uniform Data Transfer, virtually all standard protocols for data transfer were quite weak at describing the data being transferred and usually required the exchange to occur through global memory. This was especially true on Microsoft Windows: the format was described by a single 16-bit "clipboard format" and the medium was always global memory.

The problem with the "clipboard format" is that it can only describe the structure of the data, that is, identify the layout of the bits. For example, the format CF_TEXT describes ASCII text. CF_BITMAP describes a device-dependent bitmap of so many colors and such and such dimensions, but was incapable of describing the actual device it depends upon. Furthermore, none of these formats gave any indication of what was actually in the data such as the amount of detail—whether a bitmap or metafile contained the full image or just a thumbnail sketch.

The problem with always using global memory as a transfer medium is apparent when large amounts of data are exchanged. Unless you have a machine with an obnoxious amount of memory, an exchange of, say, a 20MB scanned true-color bitmap through global memory is going to cause considerable swapping

Copyright © 1995 Microsoft Corporation
All Rights Reserved

to virtual memory on the disk. Restricting exchanges to global memory means that no application can choose to exchange data *on disk* when it will usually *reside on disk* even when being manipulated and will usually use virtual memory on disk anyway. It would be much more efficient to allow the source of that data to indicate that the exchange happens on disk in the first place instead of forcing 20MB of data through a virtual-memory bottleneck to just have it end up on disk once again.

Further, *latency* of the data transfer is sometimes an issue, particularly in network situations. One often needs or wants to start processing the *beginning* of a large set of data before the end the data set has even reached the destination machine. To accomplish this, some abstraction on the medium by which the data is transferred is needed.

To solve these problems, COM defines two new data structures: FORMATETC and STGMEDIUM. FORMATETC is a better clipboard format, for the structure not only contains a clipboard format but also contains a device description, a detail description (full content, thumbnail sketch, iconic, and 'as printed'), and a flag indicating what storage device is used for a particular rendering. Two FORMATETC structures that differ only by storage medium are, for all intents and purposes, two different formats. STGMEDIUM is then the better global memory handle which contains a flag indicating the medium as well as a pointer or handle or whatever is necessary to access that actual medium and get at the data. Two STGMEDIUM structures may indicate different mediums and have different references to data, but those mediums can easily contain the exact same data.

So FORMATETC is what a consumer (client) uses to indicate the type of data it wants from a data source (object) and is used by the source to describe what formats it can provide. FORMATETC can describe virtually any data, including other objects such a monikers. A client can ask a data object for an enumeration of its formats by requesting the data object's IEnumFORMATETC interface. Instead of an object blandly stating that it has "text and a bitmap" it can say it has "A device-independent string of text that is stored in global memory" and "a thumbnail sketch bitmap rendered for a 100dpi dot-matrix printer which is stored in an IStorage object." This ability to tightly describe data will, in time, result in higher quality printer and screen output as well as more efficiency in data browsing where a thumbnail sketch is much faster to retrieve and display than a full detail rendering.

STGMEDIUM means that data sources and consumers can now choose to use the most efficient exchange medium on a per-rendering basis. If the data is so big that it should be kept on disk, the data source can indicate a disk-based medium in it's preferred format, only using global memory as a backup if that's all the consumer understands. This has the benefit of using the *best* medium for exchanges as the default, thereby improving overall performance of data exchange between applications—if some data is already on disk, it does not even have to be loaded in order to send it to a consumer who doesn't even have to load it upon receipt. *At worst*, COM's data exchange mechanisms would be *as good as anything available today* where all transfers restricted to global memory. *At best*, data exchanges can be effectively instantaneous even for large data.

Note that two potential storage mediums that can be used in data exchange are storage objects and stream objects. Therefore Uniform Data Transfer as a technology itself builds upon the Persistent Storage technology as well as the basic COM foundation. Again, this enables each piece of code in an application to be leveraged elsewhere.

1.8.3Data Selection

A data object can vary to a number of degrees as to what exact data it can exchange through the IDataObject interface. Some data objects, such as those representing the clipboard or those used in a drag & drop operation, statically represent a specific selection of data in the source, such as a range of cells in a spreadsheet, a certain portion of a bitmap, or a certain amount of text. For the life of such static data objects, the data underneath them does not change.

Other types of data objects, however, may support the ability to dynamically change their data set. This ability, however, is not represented through the IDataObject interface itself. In other words, the data object has to implement some *other* interface to support dynamic data selection. An example of such objects are those that support OLE for Real-Time Market Data (WOSA/XRT) specification.[15] OLE for Real-Time Market Data uses a data object and the IDataObject interface for exchange of data, but use the IDispatch

[15] OLE for Real-Time Market Data was formerly called the "WOSA Extensions for Real Time Market Data". More information on this and other industry specific extensions to OLE is available from Microsoft.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

interface from OLE Automation to allow consumers of the data to dynamically instruct the data object to change its working set. In other words, the OLE Automation technology (built on COM but not part of COM itself) allows the consumer to identify the specific market issues and the information on those issues (high, low, volume, etc.) that it wants to obtain from the data object. In response, the data object internally determines where to retrieve that data and how to watch for changes in it. The data object then notifies the consumer of changes in the data through COM's Notification mechanism.

1.8.4Notification

Consumers of data from an external source might be interested in knowing when data in that source changes. This requires some mechanism through which a data object itself asynchronously notifies a client connected to it of just such an event at which point a client can remember to ask for an updated copy of the data when it later needs such an update.

COM handles notifications of this kind through an object called an *advise sink* which implements an interface called IAdviseSink.[16] This sink is a body that absorbs asynchronous notifications from a data source. The advise sink object itself, and the IAdviseSink interface is implemented by the consumer of data which then hands an IAdviseSink pointer to the data object in question. When the data object detects a change, it then calls a function in IAdviseSink to notify the consumer as illustrated in Figure 2-14.

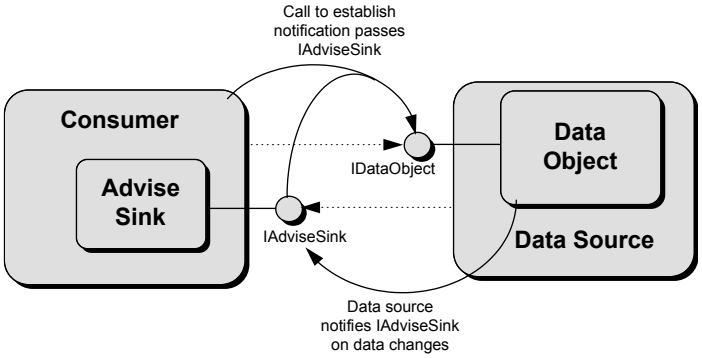


Figure 2-14: A consumer of data implements an object with the *IAdviseSink* interface through which data objects notify that consumer of data changes.

This is the most frequent situation where a client of one object, in this case the consumer, will itself implement an object to which the data object acts as a client itself. Notice that there are no circular reference counts here: the consumer object and the advise sink have different COM object identities, and thus separate reference counts. When the data object needs to notify the consumer, it simply calls the appropriate member function of IAdviseSink.

So IAdviseSink is more of a central collection of notifications of interest to a number of other interfaces and scenarios outside of IDataObject and data exchange. It contains, for example, a function for the event of a 'view' change, that is, when a particular view of data changes without a change in the underlying data. In addition, it contains functions for knowing when an object has saved itself, closed, or been renamed. All of these other notifications are of particular use in compound document scenarios and are used in OLE, but not COM proper. Chapter 14 will describe these functions but the mechanisms by which they are called are not part of COM and are not covered in this specification. Interested readers should refer to the OLE 2 Specifications from Microsoft.

Finally, data objects can establish notifications with multiple advise sinks. COM provides some assistance for data objects to manage an arbitrary number of IAdviseSink pointers through which the data object can pass each pointer to COM and then tell COM when to send notifications. COM in turn notifies all the advise sinks it maintains on behalf of the data object.

[16] Astute readers will wonder why Uniform Data Transfer is defined using the Connectable Objects interfaced described previously. The reason is simple: UDT was designed as part of the original OLE 2.0 specification in 1991, and Connectable Objects were not introduced until the release of the OLE Controls specification in 1993.

Copyright © 1995 Microsoft Corporation
All Rights Reserved