1Objects And Interfaces

This chapter describes in detail the heart of COM: the notion of interfaces and their relationships to the objects on which they are implemented. More specifically, this chapter covers what an interface is (technically), interface calling conventions, object and interface identity, the fundamental interface called IUnknown, and COM's error reporting mechanism. In addition, this chapter describes how an object implements one or more interfaces as well as a special type of object called the "enumerator" which comes up in various contexts in COM.

As described in Chapters 1 and 2, the COM Library provides the fundamental implementation locator services to clients and provides all the necessary glue to help clients communicate transparently with object regardless of where those objects execute: in-process, out-of-process, or on a different machine entirely. All servers expose their object's services through interfaces, and COM provides implementations of the "proxy" and "stub" objects that make communication possible between processes and machines where RPC is necessary.

However, as we'll see in this chapter and those that follow, the COM Library also provides fundamental API functions for both clients and servers or, in general, any piece of code that uses COM, application or not. These API functions will be described in the context of where other applications or DLLs use them. A COM implementor reading this document will find the specifications for each function offset clearly from the rest of the text. These functions are implemented in the COM Library to standardize the parts of this specification that applications should not have to implement nor would want to implement. Through the services of the COM Library, all clients can make use of all objects in all servers, and all servers can expose their objects to all clients. Only by having a standard is this possible, and the COM Library enforces that standard by doing most of the hard work.

Not all the COM Library functions are truly fundamental. Some are just convenient wrappers to common sequences of other calls, sometimes called "helper functions." Others exist simply to maintain global lists for the sake of all applications. Others just provide a solid implementation of functions that could be implemented in every application, but would be tedious and wasteful to do so.

1.1Interfaces

An interface, in the COM definition, is a contract between the user, or client, of some object and the object itself. It is a promise on the part of the object to provide a certain level of service, of functionality, to that client. Chapters 1 and 2 have already explained why interfaces are important COM and the whole idea of an object model. This chapter will now fill out the definition of an interface on the technical side.

1.1.1The Interface Binary Standard

Technically speaking, an interface is some data structure that sits between the client's code and the object's implementation through which the client requests the object's services. The interface in this sense is nothing more than a set of member functions that the client can call to access that object implementation. Those member functions are exposed outside the object implementor application such that clients, local or remote, can call those functions.

The client maintains a pointer to the interface which is, in actuality, a pointer to a pointer to an array of pointers to the object's implementations of the interface member functions. That's a lot of pointers; to clarify matters, the structure is illustrated in Figure 3-1.

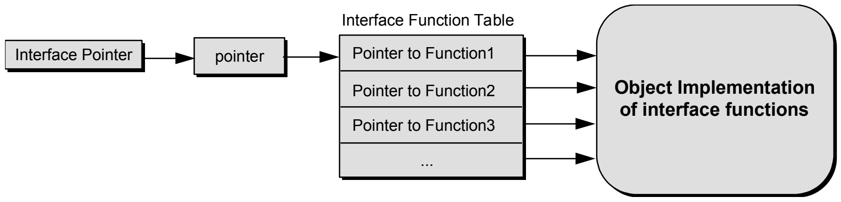


Figure 3-1: The interface structure: a client has a pointer to an interface which is a pointer to a pointer to an array (table) of pointers to the object's implementation.

By convention the pointer to the interface function table is called the pVtbl pointer. The table itself is generally referred to with the name vtbl for "virtual function table."

On a given implementation platform, a given method in a given interface (a particular IID, that is) has a fixed calling convention; this is decoupled from the implementation of the interface. In principle, this decision can be made on a method by method basis, though in practice on a given platform virtually all methods in all interfaces use the same calling convention. On Microsoft's 16-bit Windows platform, this default is the __far __cdecl calling convention; on Win32 platforms, the __stdcall calling convention is the default for methods which do not take a variable number of arguments, and __cdecl is used for those that do.

In contrast, just for note, COM API functions (not interface members) use the standard host system-call calling convention, which on both Microsoft Win16 and Win32 is the __far __pascal sequence.

Finally, and quite significantly, ***all strings passed through all COM interfaces*** (and, at least on Microsoft platforms, all COM APIs) ***are Unicode strings***. There simply is no other reasonable way to get interoperable objects in the face of (i) location transparency, and (ii) a high-efficiency object architecture that doesn't in all cases intervene system-provided code between client and server. Further, this burden is in practice not large.

When calling member functions, the caller must include an argument which is the pointer to the object instance itself. This is automatically provided in C++ compilers and completely hidden from the caller. The Microsoft Object Mapping[1] specifies that this pointer is pushed very last, immediately before the return address. The location of this pointer is the reason that the pIInterface pointer appears at the *beginning* of the argument list of the equivalent C function prototype: it means that the layout in the stack of the parameters to the C function prototype is exactly that expected by the member function implemented in C++, and so no re-ordering is required.

Usually the pointer to the interface itself is the pointer to the entire object structure (state variables, or whatever) and that structure immediately follows[2] the pVtbl pointer memory as shown in Figure 3-2.

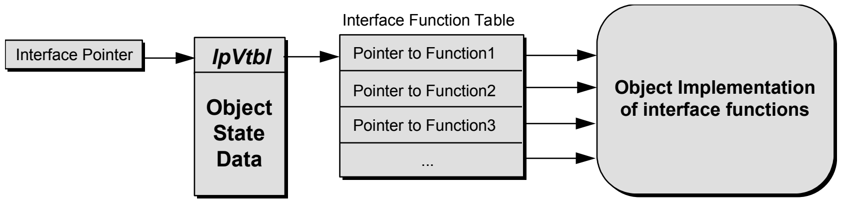


Figure 3-2: Convention places object data following the pointer to the interface function table.

Since the pVtbl is received as the this pointer in the interface function, the implementor of that function knows which object is being called—an object is, after all, some structure and functions to manipulate that structure, and the interface definition here supplies both.

In any case, this "vtbl" structure is called a binary standard because on the binary level, the structure is completely determined by the particular interface being used and the platform on which it is being invoked. It is independent of the programming language or tool used to create it. In other words, a program can be written in C to generate this structure to match what C++ does automatically. For more

[1] The "Microsoft Object Mapping" is an open specification describing the detailed layout of C++ objects. It is supported by the MS C/C++ compiler, as well as C++ compilers from other vendors including Borland, Symantec, Watcom, , and others. This is also the location of the this pointer as placed by CFront when using the traditional right-to-left __cdecl calling sequence. Thus, we achieve a large degree of interoperability.

[2] Usually this data *follows* the pVtbl pointer, but this is not required. It is perfectly legal for object-specific data to precede the vtbl pointer, and this in fact will be common with many C++ compilers.

details, see the section "C vs. C++" below. You could even create this structure in assembly if so inclined. Since compilers for other languages eventually reduce source code to assembly (as is the compiler itself) it is really a matter for compiler vendors to support this structure for languages such as Pascal, COBOL, Smalltalk, etc. Thus COM clients, objects, and servers can be written in any languages with appropriate compiler support.

Note that it is technically legal for the binary calling conventions for a given interface to vary according the particular implementation platform in question, though this flexibility should be exercised by COM system implementors only with very careful attention to source portability issues. It is the case, for example, that on the Macintosh, the pVtbl pointer does not point to the first function in the vtbl, but rather to a dummy pointer slot (which is ignored) immediately before the first function; all the function pointers are thus offset by an index of one in the vtbl.

An interface implementor is free to use the memory before and beyond the "as-specified-by-the-standard" vtbl for whatever purpose he may wish; others cannot assume anything about such memory.

1.1.2Interface Definition and Identity

Every interface has a name that serves as the programmatic compile-time type in code that uses that interface (either as a client or as an object implementor). The convention is to name each interface with a capital "I" followed by some descriptive label that indicates what functionality the interface encompasses. For example, IUnknown is the label of the interface that represents the functionality of an object when all else about that object is unknown.

These programmatic types are defined in header files provided by the designer of the interface through use of the Interface Description Language (IDL, see next section). For C++, an interface is defined as an abstract base, that is, a structure containing nothing but "pure virtual" member functions. This specification uses C++ *notation* to express the declaration of an interface. For example, the IUnknown interface is declared as:

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

where "virtual" and "=0" describe the attribute of a "pure virtual" function and where the interface keyword is defined as:

```
#define   interface   struct
```

The programmatic name and definition of an interface defines a type such that an application can declare a pointer to an interface using standard C++ syntax as in IUnknown *.

In addition, this specification as a notation makes some use of the C++ reference mechanism in parameter passing, for example:

```
QueryInterface(const IID& iid, void**ppv);
```

Usually "const <type>&" is written as "REF<type>" as in REFIID for convenience. As you might expect, this example would appear in a C version of the interface as a parameter of type:

```
const IID * const
```

Input parameters passed by reference will themselves be const, as shown here. In-out or out- parameters will not.

The use of the interface keyword is more a documentation technique than any requirement for implementation. An interface, as a binary standard, is definable in any programming language as shown in the previous section. This specification's use of C++ syntax is just a convenience.[3] Also, for ease of reading, this specification generally omits parameter types in code fragments such as this but does document those parameters and types fully with each member function. Types do, of course, appear in header files with interfaces.

It is very important to note that the programmatic name for an interface is only a *compile-time* type used in application source code. Each interface must also have a *run-time* identifier. This identifier enables a

[3] And, indeed, this syntax will at times be somewhat abused.

caller to query (via QueryInterface) an object for a desired interface. Interface identifiers are GUIDs, that is, globally-unique 16 byte values, of type IID. The person who defines the interface allocates and assigns the IID as with any other GUID, and he informs others of his choice at the same time he informs them of the interface member functions, semantics, etc. Use of a GUID for this purpose guarantees that the IID will be unique in all programs, on all machines, for all time, the run-time identifier for a given interface will in fact have the same 16 byte value.

Programmers who define interfaces convey the interface identifier to implementors or clients of that interface along with the other information about the interface (in the form of header files, accompanying semantic documentation, etc.). To make application source code independent of the representation of particular interface identifiers, it is standard practice that the header file defines a constant for each IID where the symbol is the name of the interface prefixed with "IID_" such that the name can be derived algorithmically. For example, the interface IUnknown has an identifier called IID_IUnknown.

For brevity in this specification, this definition will not be repeated with each interface, though of course it is present in the COM implementation.

1.1.3Defining Interfaces: IDL

The Interface Description Language (IDL) is based on the Open Software Foundation (OSF) Distributed Computing Environment (DCE) specification for describing interfaces, operations, and attributes to define remote procedure calls. COM extends the IDL to support distributed objects.

A designer can define a new custom interface by writing an interface definition file. The interface definition file uses the IDL to describe data types and member functions of an interface. The interface definition file contains the information that defines the actual contract between the client application and server object. The interface contract specifies three things:

- *Language binding*—defines the programming model exposed to the application program using a particular programming language.
- *Application binary interface*—specifies how consumers and providers of the interface interoperate on a particular target platform.
- *Network interface*—defines how client applications access remote server objects via the network.

After completing the interface definition file, the programmer runs the IDL compiler to generate the interface header and the source code necessary to build the interface proxy and interface stub that the interface definition file describes. The interface header file is made available so client applications can use the interface. The interface proxy and interface stub are used to construct the proxy and stub DLLs. The DLL containing the interface proxy must be distributed with all client applications that use the new interface. The DLL containing the interface stub must be distributed with all server objects that provide the new interface.

It is important to note that the IDL is a tool that makes the job of defining interfaces easier for the programmer, and is one of possibly many such tools. It is not the key to COM interoperability. COM compliance does not require that the IDL compiler be used. However, as IDL is broadly understood and used, it provides a convenient means by which interface specifications can be conveyed to other programmers.

1.1.4C vs. C++ vs. ...

This specification documents COM interfaces using C++ syntax as a notation but (again) does not mean COM requires that programmers use C++, or any other particular language. COM is based on a *binary* interoperability standard, rather than a *language* interoperability standard. Any language supporting "structure" or "record" types containing double-indirected access to a table of function pointers is suitable.

However, this is not to say all languages are created equal. It is certainly true that since the binary vtbl standard is exactly what most C++ compilers generate on PC and many RISC platforms, C++ is a *convenient* language to use over a language such as C.

That being said, COM can declare interface declarations for both C++ and C (and for other languages if the COM implementor desires). The C++ definition of an interface, which in general is of the form:

```
interface ISomeInterface
    {
    virtual RET_T  MemberFunction(ARG1_T arg1, ARG2_T arg2 /*, etc */);
    [Other member functions]
    ...
    };
```

then the corresponding C declaration of that interface looks like

```
typedef struct ISomeInterface
    {
    ISomeInterfaceVtbl *  pVtbl;
    } ISomeInterface;

typedef struct ISomeInterfaceVtbl ISomeInterfaceVtbl;

struct ISomeInterfaceVtbl
    {
    RET_T (*MemberFunction)(ISomeInterface * this, ARG1_T arg1,
        ARG2_T arg2 /*, etc */);
    [Other member functions]
    } ;
```

This example also illustrates the algorithm for determining the signature of C form of an interface function given the corresponding C++ form of the interface function:

- Use the same argument list as that of the member function, but add an initial parameter which is the pointer to the interface. This initial parameter is a pointer to a C type of the same name as the interface.
- Define a structure type which is a table of function pointers corresponding to the vtbl layout of the interface. The name of this structure type should be the name of the interface followed by "Vtbl." Members in this structure have the same names as the member functions of the interface.

The C form of interfaces, when instantiated, generates exactly the same binary structure as a C++ interface does when some C++ class inherits the function signatures (but no implementation) from an interface and overrides each virtual function.

These structures show why C++ is more convenient for the object implementor because C++ will automatically generate the vtbl and the object structure pointing to it in the course of instantiating an object. A C object implementor must define and object structure with the pVtbl field first, explicitly allocate both object structure and interface Vtbl structure, explicitly fill in the fields of the Vtbl structure, and explicitly point the pVtbl field in the object structure to the Vtbl structure. Filling the Vtbl structure need only occur once in an application which then simplifies later object allocations. In any case, once the C program has done this explicit work the binary structure is indistinguishable from what C++ would generate.

On the client side of the picture there is also a small difference between using C and C++. Suppose the client application has a pointer to an ISomeInterface on some object in the variable *psome*. If the client is compiled using C++, then the following line of code would call a member function in the interface:

```
psome->MemberFunction(arg1, arg2, /* other parameters */);
```

A C++ compiler, upon noting that the type of psome is an ISomeInterface * will know to actually perform the double indirection through the hidden pVtbl pointer and will remember to push the psome pointer itself on the stack so the implementation of MemberFunction knows which object to work with. This is, in fact, what C++ compilers do for any member function call; C++ programmers just never see it.

What C++ actually does is be expressed in C as follows:

```
psome->lpVtbl->MemberFunction(psome, arg1, arg2, /* other parameters */);
```

This is, in fact, how a client written in C would make the same call. These two lines of code show why C++ is more convenient—there is simply less typing and therefore fewer chances to make mistakes. The resulting source code is somewhat cleaner as well. The key point to remember, however, is that *how the client calls an interface member depends solely on the language used to implement the client and is completely unrelated to the language used to implement the object*. The code shown above to call an interface function is the code necessary to work with the interface binary standard and not the object itself.

1.1.5Remoting Magic Through Vtbls

The double indirection of the *vtbl* structure has an additional, indeed enormous, benefit: the pointers in the table of function pointers do not need to point directly to the real implementation in the real object. This is the heart of Location Transparency.

It is true that in the in-process server case, where the object is loaded directly into the client process, the function pointers in the table are, in fact, the actual pointers to the actual implementation. So a function call from the client to an interface member directly transfers execution control to the interface member function.

However, this cannot possibly work for local, let alone remote, object, because pointers to memory are absolutely not sharable between processes. What must still happen to achieve transparency is that the client continues to call interface member functions *as if it were calling the actual implementation*. In other words, the client uniformly transfers control to some object's member function by making the call.

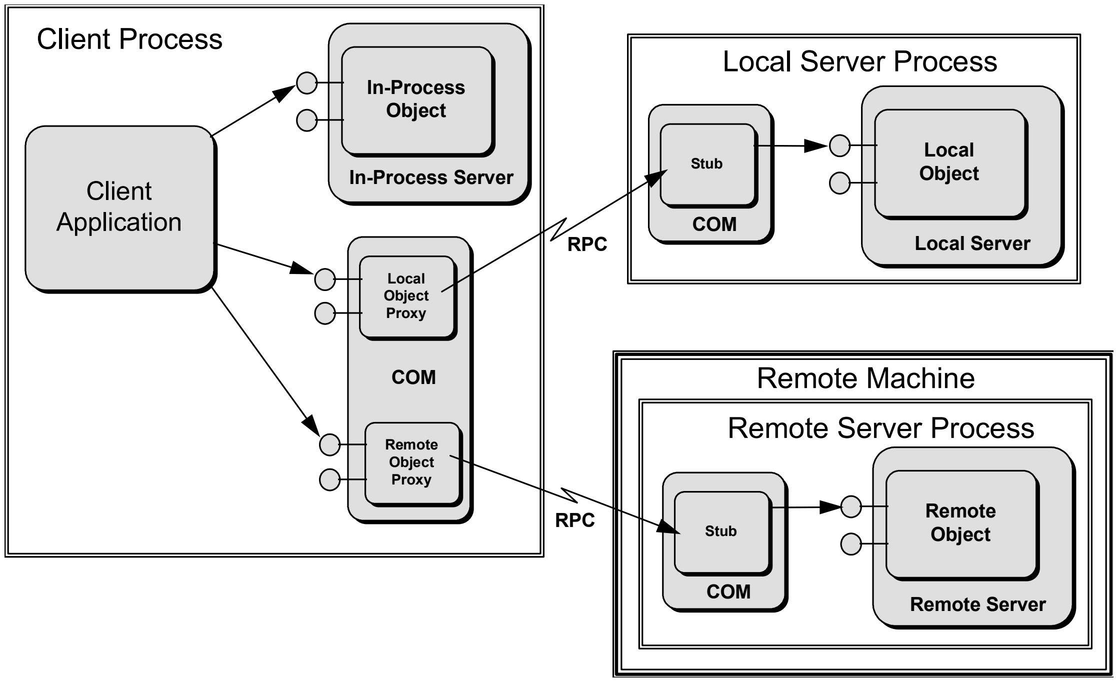


Figure 3-3: A client always calls interface members in some in-process object. If the actual object is local or remote, the call is made to a proxy object which then makes a remote procedure call to the actual object.

So what member function actually executes? The answer is that the interface member called is implemented by a proxy object that is always an in-process object that acts on behalf of the object being called. This proxy object knows that the actual object is running in a local or remote server and so it must somehow make a remote procedure call, through a standard RPC mechanism, to that object as shown in Figure 3-3.

The proxy object packages up the function parameters in some data packets and generates an RPC call to the local or remote object. That packet is picked up by a stub object in the server's process, on the local or a remote machine, which unpacks the parameters and makes the call to the real implementation of the member function. When that function returns, the stub packages up any out-parameters and the return value, sends it back to the proxy, which unpacks them and returns them to the original client. For exact details on how the proxy-stub and RPC mechanisms work, see Chapter 7.

The bottom line is that client and server always talk to each other as if everything was in-process. All calls from the client and all calls to the server do at some point, in fact, happen in-process. But because the vtbl structure allows some agent, like COM, to intercept all function calls and all returns from functions, that agent can redirect those calls to an RPC call as necessary. All of this is completely transparent to the client and server, hence Location Transparency.[4]

1.2Globally Unique Identifiers

As mentioned earlier in this document, the GUID, from which are also obtained CLSID, IIDs, and any other needed unique identifier, is a 128-bit, or 16-byte, value. The term GUID as used in this specification is completely synonymous and interchangeable with the term "UUID" as used by the DCE RPC architecture; they are indeed one and the same notion. In binary terms, a GUID is a data structure defined as follows, where DWORD is 32-bits, WORD is 16-bits, and BYTE is 8-bits:

```
typedef struct GUID {
    DWORD  Data1;
    WORD   Data2;
    WORD   Data3;
    BYTE   Data4[8];
    } GUID;
```

This structure provides applications with some way of addressing the parts of a GUID for debugging purposes, if necessary. This information is also needed when GUIDs are transmitted between machines of different byte orders.

For the most part, applications never manipulate GUIDs directly—they are almost always manipulated either as a constant, such as with interface identifiers, or as a variable of which the absolute value is unimportant. For example, a client might enumerate all object classes registered on the system and display a list of those classes to an end user. That user selects a class from the list which the client then maps to an absolute CLSID value. The client does not care what that value is—it simply knows that it uniquely identifies the object that the user selected.

The GUID design allows for coexistence of several different allocation technologies, but the one by far most commonly used incorporates a 48-bit machine unique identifier together with the current UTC time and some persistent backing store to guard against retrograde clock motion. It is in theory capable of allocating GUIDs at a rate of 10,000,000 per second per machine for the next 3240 years, enough for most purposes.

For further information regarding GUID allocation technologies, see pp585-592 of [CAE RPC].[5]

1.3The IUnknown Interface

This specification has already mentioned the IUnknown interface many times. It is the fundamental interface in COM that contains basic operations of not only all objects, but all interfaces as well: reference counting and QueryInterface. All interfaces in COM are polymorphic with IUnknown, that is, if you look at the first three functions in any interface you see QueryInterface, AddRef, and Release. In other words, IUnknown is base interface from which all other interfaces inherit.

Any single object usually only requires a single implementation of the IUnknown member functions. This means that by virtue of implementing any interface on an object you completely implement the IUnknown functions. You do not generally need to explicitly inherit from nor implement IUnknown as its own interface: when queried for it, simply typecast another interface pointer into an IUnknown* which is entirely legal with polymorphism.

In some specific situations, more notably in creating an object that supports aggregation, you may need to implement one set of IUnknown functions for all interfaces as well as a stand-alone IUnknown interface. The reasons and techniques for this are described in the "Object Reusability" section of Chapter 6.

In any case, any object implementor will implement IUnknown functions, and we are now in a position to look at them in their precise terms.

[4] Of course, if a client timed the call it might be able to discern a performance penalty if it had both in-process and out-of-process objects to compare.

[5] Though be aware that the use of the term GUID on page 587 is regrettably *not* the same as its usage in this specification. In this specification, the term GUID is used to refer to all identifiers that are "interoperable" with UUIDs as defined on p586; p587 uses the term to refer to one specific central-authority allocation scheme. Apologies to those who may be confused by this state of affairs.

1.3.1IUnknown Interface

IUnknown supports the capability of getting to other interfaces on the same object through QueryInterface. In addition, it supports the management of the existence of the interface instance though AddRef and Release. The following is the definition of IUnknown using the IDL notation; for details on the syntax of IDL see Chapter 15.[6]

```
[
  object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)
]
interface IUnknown
{
     HRESULT     QueryInterface([in] REFIID iid, [out] void **ppv) ;
     ULONG       AddRef(void) ;
     ULONG       Release(void);
}
```

IUnknown::QueryInterface

HRESULT IUnknown::QueryInterface(iid, ppv)

Return a pointer within this object instance that implements the indicated interface. Answer NULL if the receiver does not contain an implementation of the interface.

It is required that any query for the specific interface IUnknown[7] always returns the *same actual pointer value*, no matter through which interface derived from IUnknown it is called. This enables the following identity test algorithm to determine whether two pointers in fact point to the same object: call Query-Interface(IID_IUnknown, ...) on both and compare the results.

In contrast, queries for interfaces *other* than IUnknown are *not* required to return the same actual pointer value each time a QueryInterface returning one of them is called. This, among other things, enables sophisticated object implementors to free individual interfaces on their objects when they are not being used, recreating them on demand (reference counting is a per-interface notion, as is explained further below). This requirement is the basis for what is called *COM identity.*

It is required that the set of interfaces accessible on an object via QueryInterface be static, not dynamic, in the following precise sense.[8] Suppose we have a pointer to an interface

```
ISomeInterface * psome = (some function returning an ISomeInterface *);
```

where ISomeInterface derives from IUnknown. Suppose further that the following operation is attempted:

```
IOtherInterface * pother;
HRESULT   hr;
hr=psome->QueryInterface(IID_IOtherInterface, &pother);              //line 4
```

Then, the following must be true:

- If hr==S_OK, then if the QueryInterface in "line 4" is attempted a second time from the same psome pointer, then S_OK must be answered again. This is independent of whether or not pother->Release was called in the interim. In short, if you can get to a pointer once, you can get to it again.
- If hr==E_NOINTERFACE, then if the QueryInterface in line 4 is attempted a second time from the same psome pointer, then E_NOINTERFACE must be answered again. In short, if you didn't get it the first time, then you won't get it later.

Furthermore, *QueryInterface* must be reflexive, symmetric, and transitive with respect to the set of interfaces that are accessible. That is, given the above definitions, then we have the following:

Symmetric:	psome->QueryInterface(IID_ISomeInterface, ...) must succeed
Reflexive:	If in line 4, pother was successfully obtained, then

[6] Throughout this document IDL notation is used to precisely describe interfaces and other types. The actual IDL files contain additional IDL specifies that are used by the IDL compiler to optimize the generation of marshaling code, but have no bearing on the actual interface contract.

[7] That is, a QueryInterface invocation where iid is 00000000-0000-0000-C000-000000000046.

[8] While this set of rules may seem surprising to some, they are needed in order that remote access to interface pointers can be provided with a reasonable degree of efficiency (without this, interface pointers could not be cached on a remote machine). Further, as QueryInterface forms the fundamental architectural basis by which clients reason about the capabilities of an object with which they have come in contact, stability is needed to make any sort of reasonable reasoning and capability discovery possible.

	`pother->QueryInterface(IID_ISomeInterface, ...)` must succeed.
Transitive:	If in line 4, pother was successfully obtained, and we do `IYetAnother * pyet;` `pother->QueryInterface(IID_IYetAnother, &pyet); //Line 7` and pyet is successfully obtained in line 7, then `pyet->QueryInterface(IID_ISomeInterface, ...)` must succeed.

Here, "must succeed" means "must succeed barring catastrophic failures." As was mentioned above, it is specifically *not* the case that two QueryInterface calls on the same pointer asking for the same interface must succeed and return exactly the same *pointer value* (except in the IUnknown case as described previously).

Argument	Type	Description
iid	REFIID	The interface identifier desired.
ppv	void**	Pointer to the object with the desired interface. In the case that the interface is not supported or another error occurred, *ppv must be set to NULL.

Return Value	Meaning
S_OK	Success. The interface is supported
E_NOINTERFACE	The interface is not supported
E_UNEXPECTED	An unknown error occurred.

IUnknown::AddRef

ULONG IUnknown::AddRef(void)

Increments the reference count in this interface instance.

Objects implementations are required to support a certain minimum size for the counter that is internally maintained by AddRef. In short, this counter must be at least 31 bits large. The precise rule is that the counter must be large enough to support 2^{31}-1 outstanding pointer references to all the interfaces on a given object taken as a whole. Just make it a 32 bit unsigned integer, and you'll be fine.

Argument	Type	Description
return value	ULONG	The resulting value of the reference count. This value is returned solely for diagnostic/testing purposes; it absolutely holds no meaning for release code since in certain situations it is unstable

IUnknown::Release

ULONG IUnknown::Release(void)

Release a reference to this interface instance.

If AddRef has been called on this object (through any IUnknown members of its interfaces) *n* times and this is the *n*th call to Release, then the interface instance will free itself.

Release cannot indicate failure; if a client needs to know that resources have been freed etc., it must use a method in some interface on the object with higher level semantics before calling release.

Argument	Type	Description
return value	ULONG	The resulting value of the reference count. This value is returned solely for diagnostic/testing purposes; it only has meaning when the return is zero meaning that the object cannot be considered valid in any way by the caller. Non-zero values are meaningless to the caller.

1.3.2Reference Counting

Objects accessed through interfaces use a reference counting mechanism to ensure that the lifetime of the object includes the lifetime of references to it. This mechanism is adopted so that independent components can obtain and release access to a single object, and not have to coordinate with each other over the lifetime management. In a sense, the object provides this management, so long as the client components conform to the rules. Within a single component that is completely under the control of a single development organization, clearly that organization can adopt whatever strategy it chooses. The following rules are about how to manage and communicate interface instances between components, and are a reasonable starting point for a policy within a component.

Note that the reference counting paradigm applies only to pointers to interfaces; pointers to data are not referenced counted.

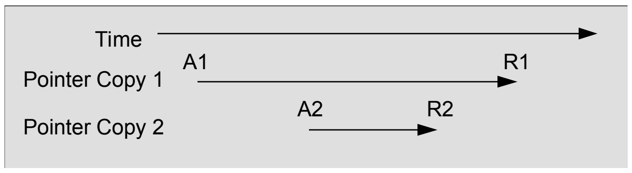
It is important to be very clear on exactly when it is necessary to call AddRef and Release through an interface pointer. By its nature, pointer management is a cooperative effort between separate pieces of code, which must all therefore cooperate in order that the overall management of the pointer be correct. The following discussion should hopefully clarify the rules as to when AddRef and Release need to be called in order that this may happen. Some special reference counting rules apply to objects which are aggregated; see the discussion of aggregation in Chapter 6.

The conceptual model is the following: interface pointers are thought of as living in pointer variables, which for the present discussion will include variables in memory locations and in internal processor registers, and will include both programmer- and compiler-generated variables. In short, it includes all internal computation state that holds an interface pointer. Assignment to or initialization of a pointer variable involves creating a *new copy* of an already existing pointer: where there was one copy of the pointer in some variable (the value used in the assignment/initialization), there is now two. An assignment to a pointer variable *destroys* the pointer copy presently in the variable, as does the destruction of the variable itself (that is, the scope in which the variable is found, such as the stack frame, is destroyed).

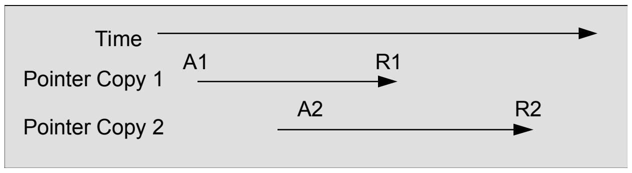
> **Rule 1**: AddRef must be called for every new copy of an interface pointer, and Release called every destruction of an interface pointer except where subsequent rules explicitly permit otherwise.

This is the default case. In short, unless special knowledge permits otherwise, the worst case must be assumed. The exceptions to Rule 1 all involve knowledge of the relationships of the lifetimes of two or more copies of an interface pointer. In general, they fall into two categories.[9]

[9] There are in fact more general cases than illustrated here involving n-way rather than 2-way interactions of matched AddRef / Release pairs, but that will not be elaborated on here.



Category 1. Nested lifetimes



Category 2. Staggered overlapping lifetimes

In Category 1 situations, the AddRef A2 and the Release R2 can be omitted, while in Category 2, A2 and R1 can be eliminated.

Rule 2: Special knowledge on the part of a piece of code of the relationships of the beginnings and the endings of the lifetimes of two or more copies of an interface pointer can allow AddRef/Release pairs to be omitted.

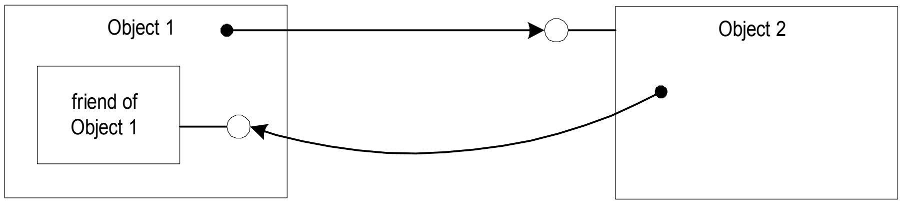
The following rules call out specific common cases of Rule 2. The first two of these rules are particularly important, as they are especially common.

Rule 2a: *In-parameters to functions.* The copy of an interface pointer which is passed as an actual parameter to a function has a lifetime which is nested in that of the pointer used to initialize the value. The actual parameter therefore need not be separately reference counted.

Rule 2b: *Out-parameters from functions, including return values.* This is a Category 2 situation. In order to set the out parameter, the function itself by Rule 1 must have a stable copy of the interface pointer. On exit, the responsibility for releasing the pointer is transferred from the callee to the caller. The out-parameter thus need not be separately reference counted.

Rule 2c: *Local variables.* A function implementation clearly has omniscient knowledge of the lifetimes of each of the pointer variables allocated on the stack frame. It can therefore use this knowledge to omit redundant AddRef/Release pairs.

Rule 2d: *Backpointers*. Some data structures are of the nature of containing two components, A and B, each with a pointer to the other. If the lifetime of one component (A) is known to contain the lifetime of the other (B), then the pointer from the second component back to the first (from B to A) need not be reference counted. Often, avoiding the cycle that would otherwise be created is important in maintaining the appropriate deallocation behavior. However, such non-reference counted pointers should be used *with extreme caution.*In particular, as the remoting infrastructure cannot know about the semantic relationship in use here, such backpointers cannot be remote references. In almost all cases, an alternative design of having the backpointer refer a second “friend” object of the first rather than the object itself (thus avoiding the circularity) is a superiour design. The following figure illustrates this concept.[10]



The following rules call out common non-exceptions to Rule 1.

Rule 1a: *In-Out-parameters to functions.* The caller must AddRef the actual parameter, since it will be Released by the callee when the out-value is stored on top of it.

[10] The connection point interfaces introduced in the OLE Controls specification are a real world example of this concept.

Rule 1b: *Fetching a global variable.* The local copy of the interface pointer fetched from an existing copy of the pointer in a global variable must be independently reference counted since called functions might destroy the copy in the global while the local copy is still alive.

Rule 1c: *New pointers synthesized out of "thin air."* A function which synthesizes an interface pointer using special internal knowledge rather than obtaining it from some other source must do an initial AddRef on the newly synthesized pointer. Important examples of such routines include instance creation routines, implementations of IUnknown::QueryInterface, etc.

Rule 1d: *Returning a copy of an internally stored pointer.* Once the pointer has been returned, the callee has no idea how its lifetime relates to that of the internally stored copy of the pointer. Thus, the callee must call AddRef on the pointer copy before returning it.

Finally, when implementing or using reference counted objects, a technique sometimes termed "artificial reference counts" sometimes proves useful. Suppose you're writing the code in method Foo in some interface IInterface. If in the implementation of Foo you invoke functions which have even the remotest chance of decrementing your reference count, then such function may cause you to release before it returns to Foo. The subsequent code in Foo will crash.

A robust way to protect yourself from this is to insert an AddRef at the beginning of Foo which is paired with a Release just before Foo returns:

```
void IInterface::Foo(void) {
    this[11]->AddRef();
    /*
     * Body of Foo, as before, except short-circuit returns
     * need to be changed.
     */
    this->Release();
    return;
    }
```

These "artificial" reference counts guarantee object stability while processing is done.

1.4Error Codes and Error Handling

COM interface member functions and COM Library API functions use a specific convention for error codes in order to pass back to the caller both a useful return value and along with an indication of status or error information. For example, it is highly useful for a function to be capable of returning a Boolean result (true or false) as well as indicate failure or success—returning true and false means that the function executed successfully, and true or false is the answer whereas an error code indicates the function failed completely.

But before we get into error handling in COM, we'll first take a small digression. Many readers might here be wondering about exceptions. How do exceptions relate to interfaces? In short, *it is strictly illegal to throw an exception across an interface invocation*; all such cross-interface exceptions which are thrown are in fact bugs in the offending interface implementation. Why have such a policy? The first, straightforward, pragmatic reason is the technical reality that there simply isn't an ubiquitous exception model or semantic that is broadly supported across languages and operating systems that one could choose to permit; recall that location transparency and language independence are important design goals of COM. Further, simplicity is also an important design goal. It is well-understood that, quite apart from COM *per se*, the exceptions that may be legally thrown from a function implementation in the public interface of an encapsulated module must necessarily from part of the contract of that function implementation. Thus, a thrown exception across such a boundary is merely an alternative mechanism by which values may be returned from the function. In COM, we instead make use of the simpler, ubiquitous, already-existing return-value mechanism for returning information from a function as our error reporting mechanism: simply returning HRESULTs, which are the topic of this section.

This all being said, it would be absolutely perfectly reasonable for the implementor of a tool for using or implementing COM interfaces to within the body of code managed by his tool turn errors returned from

[11] "This" is the appropriate thing to AddRef in an object implementation using the approach of multiply inheriting from the suite of interfaces supported by the object; more complex implementation strategies will need to modify this appropriately.

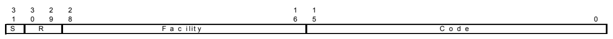
invoked COM interfaces into local exceptions and, conversely, to turn internally generated exceptions into error-returns across an interface boundary. This is yet another example of the clear architectural difference that needs to be made between the rules and design of the underlying COM system architecture and the capabilities and design freedom afforded to tools that support that architecture.

1.4.1HRESULT

The key type involved in COM error reporting is HRESULT.[12] In addition, the COM Library provides a few functions and macros to help applications of any kind deal with error information. An HRESULT is a simple 32-bit value:

```
typedef LONG HRESULT;
```

An HRESULT is divided up into an internal structure that has four fields with the following format (numbers indicate bit positions):



S: (1 bit) Severity field:

0 *Success*. The function was successful; it behaved according to its proscribed semantics.

1 *Error*. The function failed due to an error condition.

R: (2 bits) Reserved for future use; must be set to zero by present programs generating HRESULTs; present code should not take action that relies on any particular bits being set or cleared this field.

Facility: (13 bits) Indicates which group of status codes this belongs to. New facilities must be allocated by a central coordinating body since they need to be universally unique.[13] However, the need for new facility codes is very small. Most cases can and should use FACILITY_ITF. See the section "Use of FACILITY_ITF" below.

Code: (16 bits) Describes what actually took place, error or otherwise.

COM presently defines the following facility codes:

Facility Name	**Facility Value**	**Description**
FACILITY_NULL	0	Used for broadly applicable common status codes that have no specific grouping. S_OK belongs to this facility, for example.
FACILITY_ITF	4	Used for by far the majority of result codes that are returned from an interface member function. Use of this facility indicates that the meaning of the error code is defined solely by the definition of the particular interface in question; an HRESULT with exactly the same 32-bit value returned from another interface might have a different meaning
FACILITY_RPC	1	Used for errors that result from an underlying remote procedure call implementation. In general, this specification does not explicitly document the RPC errors that can be returned from functions, though they nevertheless can be returned in situations where the interface being used is in fact remoted
FACILITY_DISPATCH	2	Used for IDispatch-interface-related status codes.
FACILITY_STORAGE	3	Used for persistent-storage-related status codes. Status codes whose code (lower 16 bits) value is in the range of DOS error codes (less than 256) have the same meaning as the corresponding DOS error.
FACILITY_WIN32	7	Used to provide a means of mapping an error code from a function in the Win32 API into an HRESULT. The semantically significant part of a Win32 error is 16 bits large.
FACILITY_WINDOWS	8	Used for additional error codes from Microsoft-defined interfaces.

[12] The name "HRESULT" is retained for historical reasons. Readers familiar with programming COM on the Windows platform will note that HRESULT is analogous to SCODE.

[13] As of this writing, said body is Microsoft Corporation.

FACILITY_CONTROL	10	Used for OLE Controls-related error values.

A particular HRESULT value by convention uses the following naming structure:

<**Facility**>_<**Sev**>_<**Reason**>

where <**Facility**> is either the facility name or some other distinguishing identifier, <**Sev**> is a single letter, one of the set { S, E } indicating the severity (success or error), and <**Reason**> is a short identifier that describes the meaning of the code. Status codes from FACILITY_NULL omit the <**Facility**>_ prefix. For example, the status code E_NOMEMORY is the general out-of memory error. All codes have either S_ or E_ in them allowing quick visual determination if the code means success or failure.

The general "success" HRESULT is named S_OK, meaning "everything worked" as per the function specification. The value of this HRESULT is zero. In addition, as it is useful to have functions that can succeed but return Boolean results, the code S_FALSE is defined are success codes intended to mean "function worked and the result is false."

```
#define   S_OK       0
#define   S_FALSE    1
```

A list of presently-defined standard error codes and their semantics can be found in Appendix A.

From a general interface design perspective, "success" status codes should be used for circumstances where the consequence of "what happened" in a method invocation is most naturally understood and dealt with by client code by looking at the out-values returned from the interface function: NULL pointers, etc. "Error" status codes should in contrast be used in situations where the function has performed in a manner that would naturally require "out of band" processing in the client code, logic that is written to deal with situations in which the interface implementation truly did not behave in a manner under which normal client code can make normal forward progress. The distinction is an imprecise and subtle one, and indeed many existing interface definitions do not for historical reasons abide by this reasoning. However, with this approach, it becomes feasible to implement automated COM development tools that appropriately turn the error codes into exceptions as was mentioned above.

Interface functions in general take the form:

```
HRESULT ISomeInteface::SomeFunction(ARG1_T arg1, ... , ARGN_T argn, RET_T * pret);
```

Stylistically, what would otherwise be the return value is passed as an out-value through the last argument of the function. COM development tools which map error returns into exceptions might also consider mapping the last argument of such a function containing only one out-parameter into what the programmer sees as the "return value" of the method invocation.

The COM remoting infrastructure only supports reporting of RPC-induced errors (such as communication failures) through interface member functions that return HRESULTs. For interface member functions of other return types (e.g.: void), such errors are silently discarded. To do otherwise would, to say the least, significantly complicate local / remote transparency.

Use of FACILITY_ITF

The use of FACILITY_ITF deserves some special discussion with respect to interfaces defined in COM and interfaces that will be defined in the future. Where as status codes with other facilities (FACILITY_NULL, FACILITY_RPC, etc.) have universal meaning, status codes in FACILITY_ITF have their meaning completely determined by the interface member function (or API function) from which they are returned; *the same 32-bit value in FACILITY_ITF returned from two different interface functions may have completely different meanings*.

The reasoning behind this distinction is as follows. For reasons of efficiency, it is unreasonable to have the primary error code data type (HRESULT) be larger than 32 bits in size. 32 bits is not large enough, unfortunately, to enable COM to develop an allocation policy for error codes that will universally avoid conflict between codes allocated by different non-communicating programmers at different times in different places (contrast, for instance, with what is done with IIDs and CLSIDs). Therefore, COM structures the use of the 32 bit SCODE in such a way so as to allow the a central coordinating body[14] to define *some* universally defined error codes while at the same time allowing other programmers to define

[14] As of this writing, said body is Microsoft Corporation.

new error codes without fear of conflict by limiting the places in which those field-defined error codes can be used. Thus:

1. Status codes in facilities other than FACILITY_ITF can only be defined by the central coordinating body.
2. Status codes in facility FACILITY_ITF are defined solely by the *definer of the interface* or API by which said status code is returned. That is, in order to avoid conflicting error codes, a human being needs to coordinate the assignment of codes in this facility, and we state that he who defines the interface gets to do the coordination.

COM itself defines a number of interfaces and APIs, and so COM defines many status codes in FACILITY_ITF. By design, none of the COM-defined status codes in fact have the same value, even if returned by different interfaces, though it would have been legal for COM to do otherwise.

Likewise, it is possible (though not required) for designers of COM interface suites to coordinate the error codes across the interfaces in that suite so as to avoid duplication. The designers of the OLE 2 interface suite, for example, ensured such lack of duplication.

Thus, with regard to which errors can be returned by which interface functions, it is the case that, in the extreme,

- It is legal that any COM-defined *error* code may in fact be returned by any COM-defined interface member function or API function. This includes errors presently defined in FACILITY_ITF. Further, COM may in the future define new failure codes (but not *success* codes) that may also be so ubiquitously returned.

  Designers of interface suites may if they wish choose to provide similar rules across the interfaces in their suites.
- Further, any *error* in FACILITY_RPC or other facility, even those errors not presently defined, may be returned.

Clients must treat error codes that are unknown to them as synonymous with E_UNEXPECTED, which in general should be and is presently a legal error return value from each and every interface member function in all interfaces; interface designers and implementors *are responsible to insure* that any newly defined error codes they should choose to invent or return will be such that that existing clients with code treating generic cases as synonymous with E_UNEXPECTED this will have reasonable behavior.

In short, if you know the function you invoked, you know as a client how to unambiguously take action on any error code you receive. The interface implementor is responsible for maintaining your ability to do same.

Normally, of course, only a small subset of the COM-defined status codes will be usefully returned by a given interface function or API, but the immediately preceding statements are in fact the actual interoperability rules for the COM-defined interfaces. This specification endeavors to point out which error codes are particularly useful for each function, but code must be written to correctly handle the general rule.

The present document is, however, precise as to which *success* codes may legally be returned.

Conversely, it is *only* legal to return a status code from the implementation of an interface member function which has been sanctioned by the designer of that interface as being legally returnable; otherwise, there is the possibility of conflict between these returned code values and the codes in-fact sanctioned by the interface designer. Pay particular attention to this when propagating errors from internally called functions. Nevertheless, as noted above, callers of interfaces must to guard themselves from imprecise interface implementations by treating any otherwise unknown returned error code (in contrast with success code) as synonymous with E_UNEXPECTED: experience shows that programmers are notoriously lax in dealing with error handling. Further, given the third bullet point above, this coding practice is *required* by clients of the COM-defined interfaces and APIs. Pragmatically speaking, however, this is little burden to programmers: normal practice is to handle a few special error codes specially, but treat the rest generically.

All the COM-defined FACILITY_ITF codes will, in fact, have a *code* value which lies in the region 0x0000 — 0x01FF. Thus, while it is indeed legal for the definer of a new function or interface to make use of any codes in FACILITY_ITF that he chooses in any way he sees fit, it is highly recommended that only *code* values in the range 0x0200 — 0xFFFF be used, as this will reduce the possibility of accidental confusion with any COM-defined errors. It is also highly recommended that designers of new functions and interfaces consider defining as legal that most if not all of their functions can return the appropriate status codes defined by COM in facilities other than FACILITY_ITF. E_UNEXPECTED is a specific error code that most if not all interface definers will wish to make universally legal.

1.4.2COM Library Error-Related Macros and Functions

The following macros and functions are defined in the COM Library include files to manipulate status code values.

```
#define SEVERITY_SUCCESS            0
#define SEVERITY_ERROR              1

#define SUCCEEDED(Status)           ((HRESULT)(Status) >= 0)
#define FAILED(Status)              ((HRESULT)(Status)<0)

#define HRESULT_CODE(hr)            ((hr) & 0xFFFF)
#define HRESULT_FACILITY(hr)        (((hr) >> 16) & 0x1fff)
#define HRESULT_SEVERITY(hr)        (((hr) >> 31) & 0x1)

#define MAKE_HRESULT(sev,fac,code) \
    ((HRESULT) (((unsigned long)(sev)<<31) | ((unsigned long)(fac)<<16) | ((unsigned long)(code))) )
```

SUCCEEDED

SUCCEEDED(HRESULT Status)

The SUCCEEDED macro returns TRUE if the severity of the status code is either success or information; otherwise, FALSE is returned.

FAILED

FAILED(HRESULT Status)

The FAILED macro returns TRUE if the severity of the status code is either a warning or error; otherwise, FALSE is returned.

HRESULT_CODE

HRESULT_CODE(HRESULT hr)

HRESULT_CODE returns the error code part from a specified status code.

HRESULT_FACILITY

HRESULT_FACILITY(HRESULT hr)

HRESULT_FACILITY extracts the facility from a specified status code.

HRESULT_SEVERITY

HRESULT_SEVERITY(HRESULT hr)

HRESULT_SEVERITY extracts the severity field from the specified status code.

MAKE_HRESULT

MAKE_HRESULT(SEVERITY sev, FACILITY fac, HRESULT hr)

MAKE_HRESULT makes a new status code given a severity, a facility, and a status code.

1.5Enumerators and Enumerator Interfaces

A frequent programming task is that of iterating through a sequence of items. The COM interfaces are no exception: there are places in several interfaces described in this specification where a client of some object needs to iterate through a sequence of items controlled by the object. COM supports such enumeration through the use of "enumerator objects." Enumerators cleanly separate the caller's desire to loop over a set of objects from the callee's knowledge of how to accomplish that function.

Enumerators are just a concept; there is no actual interface called IEnumerator or IEnum or the like. This is due to the fact that the function signatures in an enumerator interface must include the type of the things that the enumerator enumerates. As a consequence, separate interfaces exist for each kind of thing that can be enumerated. However, the difference in the type being enumerated is the *only* difference between each of these interfaces; they are all used in fundamentally the same way. In other words, they are "generic" over the element type. This document describes the semantics of enumerators using a generic interface IEnum and the C++ parameterized type syntax where ELT_T, which stands for "**EL**emen**T** **T**ype"[15] is representative of the type involved in the enumeration:

```
[
   object,
   uuid(<IID_IEnum <ELT_T>>),  // IID_IEnum<ELT_T>
   pointer_default(unique)
]
interface IEnum<ELT_T> : IUnknown
{
   HRESULT Next( [in] ULONG celt, [out] IUnknown **rgelt, [out] ULONG *pceltFetched );
   HRESULT Skip( [in] ULONG celt );
   HRESULT Reset( void );
   HRESULT Clone( [out] IEnum<ELT_T>**ppenum );
}
```

A typical use of an enumerator is the following.

```
//Somewhere there's a type called "String"
typedef char * String;

//Interface defined using template syntax
typedef IEnum<char *>     IEnumString;
...
interface IStringManager {
      virtual IEnumString* EnumStrings(void) = 0;
      };
...
void SomeFunc(IStringManager * pStringMan)        {
      char *       psz;
      IEnumString * penum;
      penum=pStringMan->EnumStrings();
      while (S_OK==penum->Next(1, &psz, NULL))
            {
            //Do something with the string in psz and free it
            }
      penum->Release();
      return;
      }
```

IEnum::Next

HRESULT IEnum::Next(celt, rgelt, pceltFetched)

Attempt to get the next celt items in the enumeration sequence, and return them through the array pointed to by rgelt. If fewer than the requested number of elements remain in the sequence, then just return the remaining ones; the actual number of elements returned is passed through *pceltFetched (unless it is NULL). If the requested celt elements are in fact returned, then return S_OK; otherwise return S_FALSE. An error condition other than simply "not that many elements left" will return an SCODE which is a failure code rather than one of these two success values.

[15] "elt" by itself in the function prototypes is just "element"

To clarify:

- If S_OK is returned, then on exit the all celt elements requested are valid and returned in rgelt.
- If S_FALSE is returned, then on exit only the first *pceltFetched entries of rgelt are valid. The contents of the remaining entries in the rgelt array are indeterminate.
- If an error value is returned, then on exit no entries in the rgelt array are valid; they are all in an indeterminate state.

Argument	Type	Description
celt	ULONG	The number of elements that are to be returned.
rgelt[16]	ELT_T*	An array of size at least celt in which the next elements are to be returned.
pceltFetched	ULONG*	May be NULL if celt is one. If non-NULL, then this is set with the number of elements actually returned in rgelt.

Return Value	Meaning
S_OK	Success. The requested number of elements were returned.
S_FALSE	Success. Fewer than the requested number of elements were returned.
E_UNEXPECTED	An unknown error occurred.

IEnum::Skip

HRESULT IEnum::Skip(celt)

Attempt to skip over the next celt elements in the enumeration sequence. Return S_OK if this was accomplished, or S_FALSE if the end of the sequence was reached first.

Argument	Type	Description
celt	ULONG	The number of elements that are to be skipped.

Return Value	Meaning
S_OK	Success. The requested number of elements were skipped.
S_FALSE	Success. Some skipping was done, but the end of the sequence was hit before the requested number of elements could be skipped.
E_UNEXPECTED	An unknown error occurred.

IEnum::Reset

HRESULT IEnum::Reset(void)

Reset the enumeration sequence back to the beginning.

Note that there is no intrinsic guarantee that *exactly* the same set of objects will be enumerated the second time as was enumerated the first. Though clearly very desirable, whether this is the case or not is dependent on the collection being enumerated; some collections will simply find it too expensive to maintain this condition. Consider enumerating the files in a directory, for example, while concurrent users may be making changes.

Return Value	Meaning
S_OK	Success. The enumeration was reset to its beginning.
E_UNEXPECTED	An unknown error occurred.

IEnum::Clone

HRESULT IEnum::Clone(ppenum)

Return another enumerator which contains exactly the same enumeration state as this one. Using this function, a client can remember a particular point in the enumeration sequence, then return to it at a later time. Notice that the enumerator returned is of the same actual interface as the one which is being cloned.

Caveats similar to the ones found in IEnum::Reset regarding enumerating the same sequence twice apply here as well.

[16] Think of "rgelt" as short for "range of elt", signifying an array.

Argument	Type	Description
ppenum	IEnum<ELT_T>**	The place in which to return the clone enumerator.

Return Value	Meaning
S_OK	Success. The enumeration was reset to its beginning.
E_UNEXPECTED	An unknown error occurred.

1.6Designing and Implementing Objects

Objects can come in all shapes and sizes and applications will implement objects for various purposes with or without assigning the class a CLSID. COM servers implement objects for the sake of serving them to clients. In some cases, such as data change notification, a client itself will implement a classless object to essentially provide callback functions for the server object.

In all cases there is only one requirement for all objects: implement at least the IUnknown interface. An object is not a COM object unless it implements at least one interface which at minimum is IUnknown. Not all objects even need a unique identifier, that is, a CLSID. In fact, *only* those objects that wish to allow COM to locate and launch their implementations really need a CLSID. All other objects do not.

IUnknown implemented by itself can be useful for objects that simply represent the existence of some resource and control that resource's lifetime without providing any other means of manipulating that resource. By and large, however, most interesting objects will want to provide more services, that is, additional interfaces through which to manipulate the object. This all depends on the purpose of the object and the context in which clients (or whatever other agents) use it. The object may wish to provide some data exchange capabilities by implementing IDataObject, or may wish to indicate the contract through which it can serialize it's information by implementing one of the IPersist flavors of interfaces. If the object is a moniker, it will implement an interface called IMoniker that we'll see in Chapter 9. Objects that are used specifically for handling remote procedure calls implement a number of specialized interfaces themselves as we'll see in Chapter 7.

The bottom line is that you decide what functionality the object should have and implement the interface that represents that functionality. In some cases there are no standard interfaces that contain the desired functionality in which case you will want to design a custom interface. You may need to provide for remoting that interface as described in Chapter 7.

The following chapters that discuss COM clients and servers use as an example an object class designed to render ASCII text information from text stored in files. This object class is called "TextRender" and it has a CLSID of {12345678-ABCD-1234-5678-9ABCDEF00000}[17] defined as the symbol CLSID_TextRender in some include file. Note again that an object class does not have to have an associated CLSID. This example has one so we can use it to demonstrate COM clients and servers in Chapters 5 and 6.

The TextRender object can read and write text to and from a file, and so implements the IPersistFile interface to support those operations. An object can be initialized (see Chapter 5, "Initializing the Object") with the contents of a file through IPersistFile::Load. The object class also supports rendering the text data into straight text as well as graphically as metafiles and bitmaps. Rendering capabilities are handled through the IDataObject interface, and IDataObject::SetData when given text forms a second initializing function.[18] The operation of TextRender objects is illustrated in Figure 3-4:

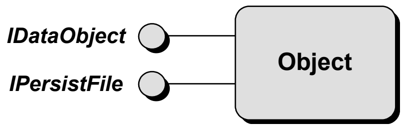


Figure 3-4: An object with *IDataObject* and *IPersistFile* Interfaces.

The "Object Reusability" section of Chapter 6 will show how we might implement this object when another object that provides some the desired functionality is available for reuse. But for now, we want to see how to implement this object on its own.

[17] Do not use this CLSID for your own purposes–it is simply an example. See the section "Identifying and Registering the Object" below.

[18] In other words, the client may initialize the object by telling it to read text from a file or by handing text to it through IDataObject::SetData. Either way, the object now has some text to render graphically or to save to a file.

1.6.1Implementing Interfaces: Multiple Inheritance

There are two different strategies for implementing interfaces on an object: multiple inheritance and interface containment. Which method works best for you depends first of all on your language of choice (languages that don't have an inheritance notion cannot support multiple inheritance, obviously) but if you are implementing an object in C++, which is a common occurrence, your choice depends on the object design itself.

Multiple inheritance works best for most objects. Declaring an object in this manner might appear as follows:

```
class CTextRender : public IDataObject, public IPersistFile {
    private:
        ULONG       m_cRef;         //Reference Count
        char *      m_pszText;      //Pointer to allocated text
        ULONG       m_cchText;      //Number of characters in m_pszText

        //Other internal member functions here

    public:
        [Constructor, Destructor]

        /*
         * We must override all interface member functions we
         * inherit to create an instantiatable class.
         */

        //IUnknown members shared between IDataObject and IPersistFile
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
        ULONG Release(void);

        //IDataObject Members overrides
        HRESULT GetData(FORAMTETC *pFE, STGMEDIUM *pSTM);
        [Other members]
        ...

        //IPersistFile Member overrides
        HRESULT Load(char * pszFile, DWORD grfMode);
        [Other members]
        ...
    };
```

This object class inherits from the interfaces it wishes to implement, declares whatever variables are necessary for maintaining the object state, and overrides all the member functions of all inherited interfaces, remembering to include the IUnknown members that are present in all other interfaces. The implementation of the single QueryInterface function of this object would use typecasts to return pointers to different vtbl pointers:

```
HRESULT CTextRender::QueryInterface(REFIID iid, void ** ppv) {
    *ppv=NULL;

    //This code assumes an overloaded == operator for GUIDs exists
    if (IID_IUnknown==iid)
        *ppv=(void *)(IUnknown *)this;

    if (IID_IPersitFile==iid)
        *ppv=(void *)(IPersistFile *)this;

    if (IID_IDataObject==iid)
        *ppv=(void *)(IDataObject *)this;

    if (NULL==*ppv)
        return E_NOINTERFACE;           //iid not supported.

    // Any call to anyone's AddRef is our own, so we can just call that directly
    AddRef();
    return NOERROR;
    }
```

This technique has the advantage that all the implementation of all interfaces is gathered together in the same object and all functions have quick and direct access to all the other members of this object. In addition, there only needs to be one implementation of the IUnknown members. However, when we deal with aggregation in Chapter 6 we will see how an object might need a *separate* implementation of IUnknown by itself.

1.6.2Implementing Interfaces: Interface Containment

There are at times reasons why you may not want to use multiple inheritance for an object implementation. First, you may not be using C++. That aside, you may want to individually track reference counts on each interface separate from the overall object for debugging or for resource management purposes—reference counting is from a client perspective an interface-specific operation. This can uncover problems in a client you might also be developing, exposing situations where the client is calling AddRef through one interface but matching it with a Release call through a different interface. The third reason that you would use a different method of implementation is when you have two interfaces with the same member function names with possibly identical function signatures or when you want to avoid function overloading. For example, if you wanted to implement IPersistFile, IPersistStorage, and IPersistStream on an object, you would have to write overloaded functions for the Load and Save members of each which might get confusing. Worse, if two interface designers should happen to define interfaces that have like-named methods with like parameter lists but incompatible semantics, such overloading isn't even possible: two separate functions need to be implemented, but C++ unifies the two method definitions. Note that as in general interfaces may be defined by independent parties that do not communicate with each other, such situations are inevitable.

The other implementation method is to use "interface implementations" which are separate C++ objects that each inherit from and implement one interface. The real object itself singly inherits from IUnknown and maintains (or contains) pointers to each interface implementation that it creates on initialization. This keeps all the interfaces separate and distinct. An example of code that uses the containment policy follows:

```
class CImplPersistFile : public IPersistFile {
    private:
        ULONG           m_cRef;             //Interface reference count for debugging

        //"Backpointer" to the actual object.
        class CTextRender * m_pObj;

    public:
        [Constructor, Destructor]

        //IUnknown members for IPersistFile
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
        ULONG Release(void);

        //IPersistFile Member overrides
        HRESULT Load(char * pszFile, DWORD grfMode);
        [Other members]
        ...
    }

class CImplDataObject : public IDataObject
    private:
        ULONG           m_cRef;             //Interface reference count for debugging

        //"Backpointer" to the actual object.
        class CTextRender * m_pObj;

    public:
        [Constructor, Destructor]

        //IUnknown members for IDataObject
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
```

```
            ULONG Release(void);

            //IPersistFile Member overrides
            HRESULT GetData(FORMATETC *pFE,STGMEDIUM *pSTM);
            [Other members]
            ...
        }


    class CTextRender : public IUnknown
        {
        friend class CImplDataObject;
        friend class CImplPersistFile;

        private:
            ULONG       m_cRef;         //Reference Count
            char *      m_pszText;      //Pointer to allocated text
            ULONG       m_cchText;      //Number of characters in m_pszText

            //Contained interface implementations
            CImplPersistFile * m_pImplPersistFile;
            CImplDataObject *  m_pImplDataObject;

            //Other internal member functions here

        public:
            [Constructor, Destructor]

            HRESULT QueryInterface(REFIID iid, void ** ppv);
            ULONG AddRef(void);
            ULONG Release(void);
        };
```

In this technique, each interface implementation must maintain a backpointer to the real object in order to access that object's variables (normally this is passed in the interface implementation constructor). This may require a *friend* relationship (in C++) between the object classes; alternatively, these friend classes can be implemented as nested classes in CTextRender.

Notice that the IUnknown member functions of each interface implementation do not need to do anything more than delegate directly to the IUnknown functions implemented on the CTextRender object. The implementation of QueryInterface on the main object would appear as follows:

```
    HRESULT CTextRender::QueryInterface(REFIID iid, void ** ppv)
        {
        *ppv=NULL;

        //This code assumes an overloaded == operator for GUIDs exists
        if (IID_IUnknown==iid)
            *ppv=(void *)(IUnknown *)this;

        if (IID_IPersitFile==iid)
            *ppv=(void *)(IPersistFile *)m_pImplPersistFile;

        if (IID_IDataObject==iid)
            *ppv=(void *)(IDataObject *)m_pImplDataObject;

        if (NULL==*ppv)
            return E_NOINTERFACE;           //iid not supported.

        //Call AddRef through the returned interface
        ((IUnknown *)ppv)->AddRef();
        return NOERROR;
        }
```

This sort of delegation structure makes it very easy to redirect each interface's IUnknown members to some other IUnknown, which is necessary in supporting aggregation as explained in Chapter 6. But overall the implementation is not much different than multiple inheritance and both methods work equally well. Containment of interface implementation is more easily translatable into C where classes simply become equivalent structures, if for any reason such readability is desirable (such as making the source code more comprehensible to C programmers who do not know C++ and do not understand multiple inheritance). In the end it really all depends upon your preferences and has no significant impact on performance nor development.

This page intentionally left blank.