

[Document title]

[Document subtitle]



[Date]

[Company name]

[Company address]

Table of Contents

No table of contents entries found.

**A com compliant object has many interesting traits.**

1. **COM is language independent:**

A com interface can be written in C++, COBOL, vb, java etc. The only true requirement that a language must satisfy is that it should be able to generate a VTABLE (binary layout) mapping to a COM object.

1. Not only Developers can write COM objects in any language, the COM objects themselves can be accessed by any COM language.

For example a JAVA application can access a COM component written in C++ or Delphi.

1. COM encapsulates the inner implementation details of the object including the

Language it was written in. All that the COM object user sees is a set of well-defined interfaces supported by the object.

1. **COM provides location transparency**

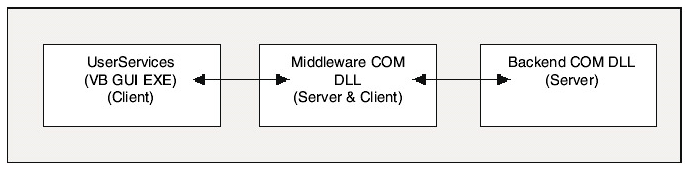
Before we understand location transparency lets add a few words to the COM vocabulary (lexicon)

1**. Process:** A process is a partition of memory containing a running application (i.e. an active main thread) along with any necessary system resources and subsidiary threads and external binaries required by the application.

2. **Client:** A client in COM is any piece of software that makes use of a COM object.

**3. Server:** In COM we define a server as a binary package (DLL or EXE) that contains one or more COM objects. Typically a single com server is known to be a home of many related COM objects, each supporting many number of interfaces.

Always keep in mind that the distinction between client and server can become blurred a little bit as it is common to have a client using a server that is a client to another server. The following diagram explains it:



Now with the three terms in place i.e. Client, Server and Process we can now distinguish between the possible relationships a client and server may have

1. An In-Process Relationship.

2. An Out of Process Relationship.

3. A Remote Relationship.

**UNDERSTANDING THE IN-PROCESS RELATIONSHIP:**

The relationship is abbreviated as in-proc relationship. In-proc servers are loaded into the same **memory partition (process)** in which the client application is running. They are in the same memory partition of the client that they are servicing.

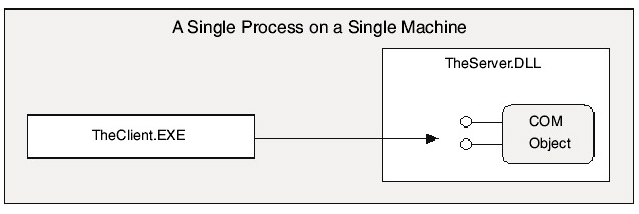
**Key Benefit:**

Speed. Since the servers are loaded into the same memory partition as client, the calls to the interfaces are as fast as making a local function call as the client and the server are **communicating through interface pointers.**

**Drawback:**

1. **Fault Tolerance**: The biggest drawback of an in-proc server is robustness or low level of fault tolerance. If the server crashes it will bring down the entire client process too.

2. **Security**: The in-process COM server will always take the security context of the Client Application they are loaded into.



**NOTE:**

COM has different threading models. IF the client and the server share different threading models then direct interface connection is not achieved.

**UNDERSTANDING THE OUT-OF-PROCESS RELATIONSHIP**

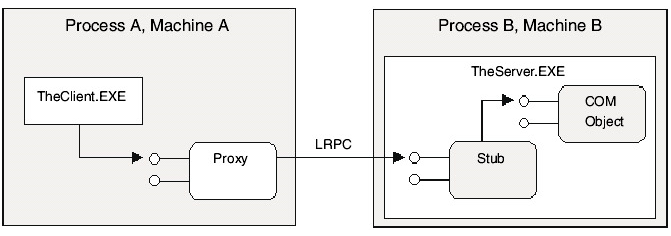
The out-of-process relationship is also termed as a local relationship. The client and the server reside on the same physical machine but they run in their own individual (distinct) memory segment as separate processes.

The two processes run with their own security contexts hence a higher level of security.

The major drawback is that it is slower as the requests have to be packaged, sent and unpackaged between process boundaries. While COM takes care of this packing and unpacking using a technique called **universal marshalling** and we rarely have to take care of it, but the process itself is time consuming.

**The major advantage** is that the system is fault tolerant as if the out of process com server crashes the client keeps running in its own process space.

The following diagram shows the out-of-process COM server and the relation between client/server.



As shown in the above example the client and the server processes communicate and send information through **LRPC (Lightweight remote procedure call).**They run on the same machine but in different processes.This is a proprietary communication protocol under the hood of the COM runtime.

In this local relationship the client and the server do not communicate directly through interface pointers but communicate using stubs and proxies. The data is transmitted between stubs and proxies.

As far as the client is concerned, the proxy is the COM object and can program against it as if it was an in-proc server. This will be discussed further.

**UNDERSTANDING THE REMOTE RELATIONSHIP**

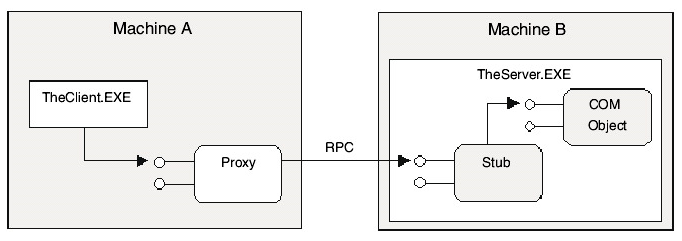
The remote COM server resides physically on any remote machine other than the machine on which the client is running.

This is a core requirement of any enterprise based solution, where the user services, business rules, and data retrieval objects can be distributed and reside on many networked computers on the network.

The remote client server COM relationships are the slowest amongst all the relationships as not only we need to package/unpackaged and marshal (assemble arrange in order) data but we also have to take into consideration the network and latency issues (Bandwidth etc.).

Remote communication is achieved through **DCOM protocol** which uses **Remote Procedure Call** mechanisms to **establish communications and forward information** between the Client and the server.

The following diagram represents a remote relationship:



**NOTE:**

The remote COM servers can be housed in DLLs or EXEs in the remote machine. Legacy COM servers can also be used and remoted using the special COM Surrogate process/utility called the dllhost.exe.

EXE server types were a “Classic” way to remote COM servers especially over LAN. This is still an effective solution.

However these days MTS is more popular. MTS is Microsoft transaction server which is a super surrogate and will host “only” COM dlls while taking care of all the details like threading concurrency, atomic transactions, and memory management and so on.

**LOCATION TRANSPARENCY**

One benefit of com states that COM provides location transparency where the client programs against the proxies as if it was making in-process function calls without knowing the location of the server.

This also states that the client side code used to access the interfaces of the server side COM object need not change based on the location of the COM object.

**Stubs and proxies** provide a level of encapsulation and the COM client treats the proxy object as a real object and can program against it as if the object was running in the clients own process space.

Likewise the COM server believes that the **STUB is the real client** leaving the server code intact.

The COM library does provide a number of methods and data structures to help optimize remote access. But these are optional. Using configuration utilities such as dcomcnfg.exe, you may redirect your client to access a server on any machine on the network without altering a single line of code.

These utilities and library extensions provide a far cleaner approach than we had before location transparency. At that time, developers needed to use different (and unrelated) APIs to move information between in-proc, local, and remote servers. As a COM developer, you do not write different code bases for client access to remote objects, local objects, or in-proc objects. Clients just see interfaces, and location transparency allows that client to be blissfully unaware of the exact location of the server.

**COM IS OBJECT ORIENTED**

Some people argue that COM is not object oriented. But it is.

* **ENCAPSULATION:**

COM does a good work in information hiding. All that is exposed to the clients are proxies. The underlying implementation is absolutely hidden from the clients.

* **POLYMORPHISM:**

As we have seen from the SHAPES example any object is allowed to implement the same interface with its own unique implementation. COM provides an ad-hoc polymorphism using the same interface.

* **INHERITANCE:**

COM does not provide a classical inheritance of binary objects. In other words we cannot say that COM object A derives from COM object B.

(For example, C3DRect used interface inheritance to bring in the definitions of IDraw and IShapeEdit). What we cannot do with COM is something along the lines of the following:

// this is not possible in COM.

// assume *BLOB* is a new keyword that defines a new binary COM object.

BLOB MyServer.NewBlob.EXE: public MyOtherServer.OldBlob.DLL

{

...

};

**COM CONTAINMENT AND AGGREGATION**

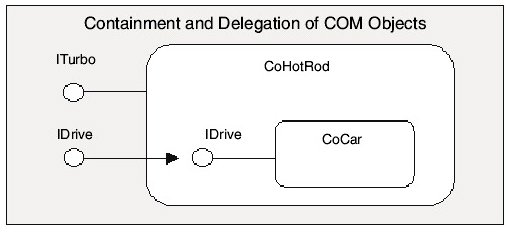
Although COM does not support inheritance it supports component reuse using Containment and Aggregation. **We can reuse behaviours (a.k.a. interfaces)** across binary COM objects via containment and aggregation. In fact on of the primary reasons COM does not support inheritance because COM was designed to keep the objects as self-sufficient as possible.

COM is a **binary standard**. So essentially a base class in **COM** will be a binary object. If we have to change something in the base class how do we know what to change? So COM side steps all these complexities by avoiding classical inheritance.

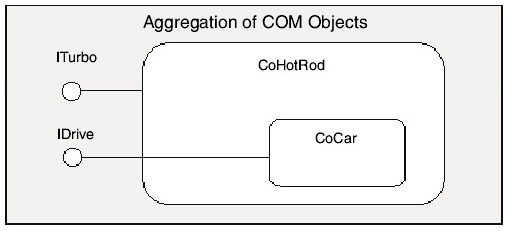
We however have binary reuse in COM using containment and delegation.

A COM object may create other COM objects and allow the access to the inner objects interfaces using delegation. Clients are unaware that the outer object (Server) is contains inner objects too. This is quite similar to out code based containment that we see in OOP where outer class contains inner class. The difference is that in the case of COM the outer class builds the inner class using the COM library.

The following figures show containment (delegation) and aggregation.



The notion of containment and delegation. Notice that CoHotRod is an "outer class" that creates and maintains an "inner class" named CoCar. CoHotRod supports the IDrive interface, which has methods that forward calls to the inner object.



COM does offer another form of binary reuse, **called *aggregation***. Here a COM object creates inner COM objects (just like the containment/delegation model). This time, however, the outer object directly extends its own set of interfaces with the interfaces of the inner objects. This gives clients the illusion that the object they are working with is composed of more interfaces than it really is, as shown above.

**In aggregation, the outer object does not "forward" calls via delegation**, as the inner object's interfaces are directly exposed as part of the outer object. We will detail the implementation of COM containment and aggregation later, so we will hold off on the specifics for now.

**COM Provides Clean Versioning of Components**

We have seen that interfaces may be safely versioned by using interface inheritance. COM uses these same versioning techniques. As we have already seen, when an object implements a given interface, it may not alter that interface in any way or else it risks breaking clients. **The COM interface is the key to robust versioning.**

**The COM, OLE, and ActiveX Relationship**

COM is the backbone of each and every ActiveX and OLE technology. Pick up your current issue of Microsoft Systems Journal, and I'd bet a pay check that at least one article discusses a new set of COM interfaces, COM service, or some other COM advancement. The point here (other than I can keep my pay check to myself) is that COM technologies are bombarding us left and right. These days, any new software item coming from those kind folks in Redmond is delivered as a set of COM extensions-not a C-based API.

Before we can really appreciate the full set of COM-based technologies examined throughout this book, a brief (and painless) history lesson is in order, beginning with OLE 1.0.

**OLE 1.0**

OLE 1.0 was a 16-bit technology that appeared on the scene circa 1991. This technology enabled software applications to share information through linking and embedding. As you may know, certain applications may export visual objects to other applications. A host application may embed this object inside itself (which produced a copy of the original data) or reference it through a link (which is a connection to the original data source). The classic example of this technology is an MS Word document containing an embedded MS Excel spreadsheet. Under the hood, OLE 1.0 was not based on the COM we know today, but rather the clunky Dynamic Data Exchange (DDE) protocol. At this time, OLE was a technology specifically used for object linking and embedding.

**OLE 2.0**

Around 1993, 16-bit OLE 2.0 was released for the Windows 3.x OS. This marked a huge directional change for OLE, as the underpinnings of DDE were beginning to be stripped away and replaced with the COM infrastructure. With the release of Windows NT 3.51, OLE 2.0 was moved into 32 bits. The major additions to the existing 16-bit technology set was the full support for Unicode string handling, as well as tweaking **COM itself to run completely on top of Microsoft's variation of the Open Software Foundation's Remote Procedure Call (OSF RPC) paradigm, rather than the older DDE paradigm.**

Linking and embedding technology was still a core service of OLE 2.0-however, it did not stop there. OLE 2.0 introduced a whole slew of COM-related technologies. The key to OLE 2.0 was the extendable COM architecture. Given this, we will never see an OLE 3.0. Instead, we'll see more COM-related technologies. **Thus, the terms OLE 1.0 and OLE 2.0 are not much more than historical footnotes in COM's family history.**

**ActiveX**

ActiveX is the current blanket name for any COM-based technology. At one time, ActiveX referred to only web-specific COM technologies; however, these days most things COM are dubbed "ActiveX" something or other (ActiveX controls, ActiveX documents, ActiveX servers, and so on). To make things a bit more confusing, Microsoft occasionally uses the legacy term "OLE" to name some newer COM-based services, such as OLE DB (which really should have been ActiveX-DB to be consistent). We will see many ActiveX services pop up throughout the remainder of this book. Just realize that all ActiveX/OLE technologies are built off the COM protocol.

**STANDARD AND CUSTOM INTERFACES IN COM**

**COM** is all about interfaces. The only way COM clients and servers communicate is through interfaces.

Each and every interface developed is developed to specify a possible behaviour some class may support. We define an interface as the following:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***An interface is a collection of semantically related functions which define a unique and single behaviour supported by the class. For example text manipulation functionality can be clubbed together as an interface called ITextFun, Graphics as IGuiFun and so on.***

This same definition works almost perfectly in the world of COM. However, COM interfaces fall under two distinct categories. Under the hood, each type of interface is exactly the same: a collection of semantically related functions—no state, no implementation.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

COM interfaces fall under two categories.

1. Standard COM Interfaces
2. Custom COM interfaces

**CUSTOM COM INTERFACES**

Custom COM interfaces are developed to solve one particular programming problem. We may create custom COM interfaces as a C++ abstract classes and extend them through numerical versioning. Every Custom interface must directly or indirectly derive from IUNKNOWN or the de-facto COM interface.

For example IDRAW can be a COM interface only if it was specified as the following:

**Interface IDRAW: public IUNKNOWN {**

**//brings in pure virtual definitions of the functions Queryinterface (), addref () and release ()**

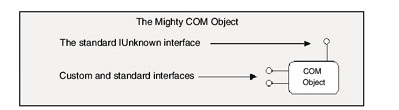
**Virtual void Draw () = 0;**

**}**

The above code suggests that the vtable of IDRAW will be coloured by the three members of IUNKNOWN (addref (), Queryinterface () and Release (). This is very true indeed. Every method of a com interface will have the three members of IUnknown as the first three members.

**Be aware that in interface inheritance, you are not required to explicitly relist inherited pure virtual methods in the derived interface. You may choose to relist the inherited pure virtual methods of IUnknown in your IDraw interface if you wish to be extremely clear; however, the compiler could care less. The class implementing IDraw would (of course) need to provide definitions for all four virtual functions**.

**Given that IUnknown is a mandatory requirement of any COM object, it holds a special place in COM notation. The IUnknown interface is always represented as the lollipop extending from the top of a COM class.**



**Standard COM Interfaces**

These are predesigned and well known COM interfaces useful in COM development. Every standard interface must be derived from IUnknown or other IUnknown derived interfaces.

***Having a set of predefined standard interface is really good as it provides a really big area of standard behaviours necessary for a component-based distributed architecture such as COM.***

In fact what makes one COM based technology different from another is the set of standard interfaces supported by the COM object.

ActiveX controls support a number of interfaces not found in automation servers.

ActiveX documents support a number of interfaces not found in OLE/COM drag and Drop.

This does not mean that the aforementioned technologies are mutually exclusive. The interfaces providing rendering, persistence and many other standard behaviour are implemented by numerous COM technologies.

**UNDERSTANDING THE INTERFACE IDENTIFIERS**

In the previous examples of interface based programming using C++ we used a custom enumeration type called INTERFACEID. This served as a way to uniquely identify an interface used in our mini system. **This technique worked fine for us as the interfaces developed were only used in the context of a single C++ application.**

In the real world COM objects can be remoted around the world! Quite Literally! Simple numerical identifiers such as {0, 1, 2, 3} etc. are bound to create name clashes (as would simple string names. Imagine the number of developers that consider IDataBase as a good name for an interface. There would be thousands!

In COM every interface should be named. GUID comes to the rescue. GUID is a 128 bit unique identifier generated based on the unique network address and the time at that point precise to the 100th nanosecond.

In COM every interface is tied to this unique GUID or 128 bit identifier. It is pronounced as GOUID.

1. When a GUID is referring to an interface identifier, we will call it as an **IID** or interface identifier.
2. COM used GUIDs to correctly identify COM classes in that case we call the **CLSIDs** or class ids.
3. COM uses GUIDs to correctly identify type libraries in that case we call them **LIBIDs**
4. COM uses GUIDs to correctly identify COM Applications or COM exes. In that case we call them **APPIds**.
5. There are numerous other COM objects.

TYPE LIBRARY

A type library (.tlb) is a binary file that stores information about a COM or DCOM object's properties and methods in a form that is accessible to other applications at runtime. Using a type library, an application or browser can determine which interfaces an object supports, and invoke an object's interface methods. This can occur even if the object and client applications were written in different programming languages. The COM/DCOM run-time environment can also use a type library to provide automatic cross-apartment, cross-process, and cross-machine marshalling for interfaces described in type libraries.

The GUID in reality is a four field structure:

**//it is represented as a structure defined in <winnt.h>**

Typedef struct\_GUID {

Unsigned long Data1;

Unsigned short Data2;

Unsigned short Data3;

Unsigned char Data4 [8];

} GUID;

The COM library gives us a set of useful functions and types for programmatically work with GUID types. These are defined in <wtypes.h>. Let’s see some of these useful functions and types:

**// <wtypes.h> lists a number of defines to programmatically work with GUID types**

#define REFGUID const GUID \* const

#define REFIID const IID \* const

#define REFCLSID const CLSID \* const

**//<objbase.h> we are also given a set of COM library functions to do comparisons of two existing GUIDS**

BOOL IsEqualGUID(REFGUID g1, REFGUID g2)

BOOL IsEqualIID(REFIID i1, REFIID i2)

BOOL IsEqualCLSID(REFCLSID c1, REFCLSID c2)

Each function performs a memcmp() of the two GUIDS(structures) and returns a true or false

BOOL IsEqualGUID (REFGUID g1, REFGUID g2)

{

Return !memcmp(g1,g2,sizeof(GUID))

}

**NOTE:** An inline version of IsEqualGUID is also provided InlineIsEqualGUID (REFGUID g1, REFGUID g2). Use with care as inline functions can bloat up the code.

In addition to IsEqualGUID () the COM library has overridden the C++ equality operator (==) and the not equal operator (! =), allowing us to compare two GUIDS in a simpler way.

The implementation internally uses IsEqualGUID function:

BOOL operator == (const GUID& g1, const GUID& g2) {

Return IsEqualGUID (g1, g2);

}

BOOL operator! = (const GUID& g1, const GUID& g2) {

Return! IsEqualGUID (g1, g2);

}

MISCELLANEOUS

**Const Correctness or use of the const keyword in C++:**

Const correctness is using the const keyword correctly is C++ to prevent the const variable/object from getting mutated. For example If we wanted to declare a function that accepts a std::string we also want to guarantee that the original value will not be changed then const keyword comes into picture.

Void fun (std:: string& mystring)

{

Mystring = “change”; //the string can be changed

}

Void fun (const std:: string& mystring)

{

Mystring = “change”; //error

}

**Some variants:**

void f1 (const std::string& s); // Pass by reference-to-const

void f2(const std::string\* sptr);  // Pass by pointer-to-const

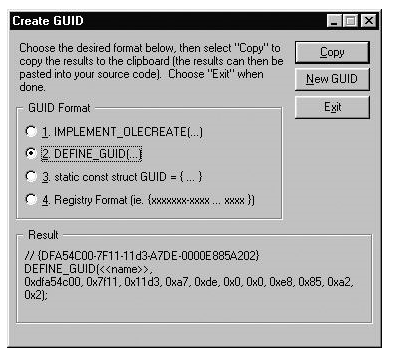
void f3(std::string s);  // Pass by value

**CREATING CUSTOM GUIDS**

Every standard COM interface has already an IID (Interface ID) defined. For example the IID of IUNKNOWN is defined as {00000000-0000-0000-C000-000000000046}

When a programmer programs a new Interface, the job of associating the new Interface to an IID is upon the programmer.

Visual studio provides a utility for that named **guidgen.exe** with a friendly user interface. A command line interface is also supported.



NOTES ABOUT GUIDGEN.EXE

1. Guidgen.exe provides four possible formats of the GUID.
   1. The first format is a MFC specific MACRO which is used to define a GUID for a MFC based class factory. For COM let us ignore this GUID format completely to avoid confusion.
   2. Format 4 is useful when we are writing an IDL(Interface Definition Language)
   3. Format 3 is what the DEFINE\_GUID macro basically expands to.
   4. We are left finally with the DEFINE\_GUID macro which we can associate a newly generated 128 bit GUID to a human readable constant. Much like the INTERFACE id enumerations we defined in the previous exercise! When we paste the newly generated GUID from the clipboard we see the following:

// {4B475690-DE06-11d2-AAF4-00A0C9312D57}

DEFINE\_GUID (<<**name**>>,

0x4b475690, 0xde06, 0x11d2, 0xaa, 0xf4, 0x0, 0xa0, 0xc9, 0x31, 0x2d, 0x57);

1. We select the **<<name>>** and replace it with a human readable constant and use it to refer to the underlying GUID.

// {4B475690-DE06-11d2-AAF4-00A0C9312D57}

DEFINE\_GUID (**IID\_IDraw**,

0x4b475690, 0xde06, 0x11d2, 0xaa, 0xf4, 0x0, 0xa0, 0xc9, 0x31, 0x2d, 0x57);

NOTE: It is a common convention to prefix a user defined interface name with IID\_ for example **IID\_MyInterface.**

**DEFINE\_GUID expands to the following macro. Notice how the long, word, and byte parameters are simply sent into the GUID fields. Also see that the name becomes the first parameter and hence the name of this newly assigned structure.**

// The DEFINE\_GUID macro creates a new GUID structure of some name.

#define **DEFINE\_GUID** (**name**, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8) \

EXTERN\_C const GUID **name** \

= {l, w1, w2, {b1, b2, b3, b4, b5, b6, b7, b8}}

In this chapter we will use the DEFINE\_GUID macro when we create new GUIDs to identify our COM items. To do so, your **projects must make a pre-processor include to the <initguid.h> header file in order to obtain the DEFINE\_GUID macro definition**. We will be creating interface header files in the upcoming labs, but just remember you can't use DEFINE\_GUID without including <initguid.h> in your project.

On a final GUID-related note, the COM library function CoCreateGuid () may be used to create a GUID programmatically. Just send in a reference to a GUID, and it will stuff the fields:

**// If you ever need to create a GUID on the fly...**

HRESULT CoCreateGuid (GUID \*pguid);

|  |  |  |
| --- | --- | --- |
|  | **Note** | Because the use of guidgen.exe is so common in COM and ATL programming, you may wish to create a custom menu item in the Visual C++ IDE to access it. To do so, select the Tools | Customize... menu selection and from the resulting dialog box, select the Tools tab. Type in a name for this custom menu item using the Menu Contents list box and map a path to guidgen.exe from the Command edit box (i.e., C:\Program Files\Microsoft Visual Studio\Common\Tools\Guidgen.exe). Select the Close button and examine your Tools menu. You will see your custom menu item is now visible and can be selected to run guidgen.exe. |

**Understanding AddRef () and Release () Rules**

Now that we understand the functionality of AddRef () and Release (), we need to understand the exact circumstances in which to call them. Proper reference counting can be broken down into the following small set of rules:

* A COM object calls AddRef () on itself whenever it successfully hands out an interface pointer to the client, typically during a Queryinterface () invocation. We will see this rule in action in the next topic section.
* When a client has successfully received an interface pointer (through a COM library call or some interface method), the client must assume that AddRef() has been called by the object, and must call Release() when finished with the acquired pointer:

// assume the following method returns an interface pointer.

IDraw\* pDraw = NULL;

if (GoGetIDraw(&pDraw)) // Success? The object has been AddRef-ed.

{

pDraw -> Draw();

pDraw -> Release(); // If this is the only reference, the object is now

// destroyed.

}

* If a client (or server for that matter) makes a copy of an interface pointer, the client should call AddRef() on that copy explicitly:

// Assume we have fetched some interfaces and now set one equal to the other.

pIFaceTwo = pIFaceOne;

pIFaceTwo -> AddRef();

Functions that take interface pointers as parameters should AddRef() and Release() the interface during the method invocation:

// This method draws any IDraw compatible object.

HRESULT DrawMe(IDraw\* pDrawingObject)

{

pDrawingObject -> AddRef(); // We are using this object.

pDrawingObject -> Draw();

pDrawingObject -> Release(); // All done.

}

At times, these calls may seem redundant, although technically correct. You can minimize AddRef () and Release () calls under a few circumstances; however, with these rules fixed in your head, you can be assured objects are deleted from memory as efficiently as possible. It is far better to add in explicit AddRef () and Release () calls (even if things seem a bit redundant) rather than have objects hanging around in memory longer than necessary.

In COM, the objects we activate with COM library calls are responsible for removing themselves from memory when they are no longer wanted. This is in contrast to our Shapes lab from the previous chapter where we indirectly deallocated our C3DRect using a custom API call (DestroyThe3DRect ()). This is not the case in COM. As long as you abide by the AddRef () and Release () rules, you can rest assured that your objects take care of themselves.

**IUnknown Establishes an Object's Identity**

You have seen that this single COM interface is responsible for the basic riggings of a COM class: object lifetime and handing out interface pointers. QueryInterface () has been defined by the COM specification with the following requirement in mind:

If a client has some interface pointer, the client must be able to navigate to any other interface on the object from that pointer.

In other words, QueryInterface () must ensure the rules of reflexivity, symmetry, and transitivity. This breaks down to the following set of guidelines:

* **Reflexivity:** If a client has an interface pointer, pA, a client must be able to query pA for itself.
* **Symmetry:**If a client can navigate from an interface pointer, pA, to another interface pointer, pB, the client must be able to navigate from pB back to pA.
* **Transitivity:**If a client can navigate from pA to pB, and from pB to pC, the client must be able to return from pC to pA.

What these rules mean to a client is simple: "**From a valid interface pointer I can get to any other interface on the object.**"

In order to ensure your **coclasses** support each of these rules, be sure each custom interface derives from an IUnknown-derived interface, and in the implementation of QueryInterface() include a provision for each and every interface on the object. Be extra careful to ensure that if your coclass supports a versioned interface (IDraw, IDraw2), a provision for each version is supported in your QueryInterface () logic, not a single test for the most derived interface.