1COM Clients

As described in earlier chapters, a COM Client is simply any piece of code that makes use of another object through that object's interfaces. In this sense, a COM Client may itself be a COM Server acting in the capacity of a client by virtue of using (or reusing) some other object.

If the client is an application, that is, an executable program as opposed to a DLL, then it must follow all the requirements for a COM Application as detailed in Chapter 4. That aside, clients have a number of ways to actually get at an object to use as discussed in a previous chapter. The client may call a specific function to create an object, it might ask an existing object to create another, or it might itself implement an object to which some other code hands yet another object's interface pointer. Not all of these objects must have CLSID.

This chapter, however, is concerned with those clients that want to create an object based on a CLSID, because at some point or another, many operations that don't directly involve a CLSID do eventually resolve to this process. For example, moniker binding internally uses a CLSID but shields clients from that fact. In any case, whatever client code uses a CLSID will generally perform the following operations in order to make use of an object:

1. Identify the class of object to use.
2. Obtain the "class factory" for the object class and ask it to create an uninitialized instance of the object class, returning an interface pointer to it.
3. Initialize the newly created object by calling an initialization member function of the "initialization interface," that is, one of a generally small set of interfaces that have such functions.
4. Make use of the object which generally includes calling QueryInterface to obtain additional working interface pointers on the object. The client must be prepared for the potential absence of a desired interface.
5. Release the object when it is no longer needed.

The following sections cover the functions and interfaces involved in each of these steps. In addition, the client may want to more closely manage the loading and unloading of server modules (DLLs or EXEs) for optimization purposes, so this chapter includes a section of such management.

As far as the client is concerned, the COM Library exists to provide fundamental implementation locator and object creation services and to handle remote procedure calls to local or remote objects (in addition to memory management services, of course). How a server facilitates these functions is the topic of Chapter 6.

Before examining the details of object creating and manipulation, realize that after the object is created and the client has its first interface pointer to that object, the *client cannot distinguish an in-process object from a local object from a remote object* by virtue of examining the interface pointer or any other interfaces on that object. That is, all objects appear identically to the client such that after creation, all requests made to the object's services are made by calling interface member functions. Period. There are not special exceptions that a client must make at run-time based on the distance of the object in question. The COM Library provides any underlying glue to insure that a call made to a local or remote object is, in fact, marshaled properly to the other process or the other machine, respectively. *This operation is transparent to the client*, who always sees any call to an object as a function call to the objects interfaces as if that object were in-process. This consistency is a key benefit for COM clients as it can treat all objects identically regardless of their actual execution context. If you are interested in understanding how this transparency is achieved, please see Chapter 7, "Communicating via Interfaces: Remoting" for more details. There you will find that all clients do, in fact, always call an in-process object first, but in local and remote cases that in-process object is just a proxy that takes care of generating a remote procedure call.

1.1Identifying the Object Class

A central feature of COM is that a client can opaquely locate and dynamically load the specific piece code that knows how to manipulate a specific class of object. This is accomplished through the COM-supplied implementation locator services through which COM associates a class identifier, that is, CLSID, with the server module for that object class. Therefore the COM Library is responsible for defining how this association occurs which usually involves a system-wide persistent registry of CLSIDs and their corresponding servers. For example, under Microsoft Windows the COM Library stores the pathnames of in-process server DLLs and local server EXEs in the system registry under the text string of the object's CLSID.

The practical upshot of all this for client applications is that the client need not know nor care how this information is maintained or how the COM Library performs the association from CLSID to server. In the same manner the client need not perform any additional work to establish communication with a local or remote object as such steps are also handled in COM transparently.

This does leave the question of how the client determines what CLSID to hand to COM in the first place. There is no single answer, for it varies from situation to situation. In some cases the object to use has a well-known and fixed CLSID that is compiled into the client application. In other cases the client may have a constant text string (compiled, that is) that represents a CLSID and uses some means to associate that name with a CLSID. Another example may be that the client has some previously saved information that directly or indirectly translates to a CLSID, such as a piece of storage (where the CLSID is serialized into a stream) or a moniker (where the CLSID is implied by the data which the moniker references). Finally, there may be some means through which the client displays a list of available objects to the end-user where each item in the list corresponds to a specific CLSID. In such cases the list is generated by browsing the registry for all existing object classes. Other examples are clearly possible, particularly in network situations.

1.2Creating the Object

Given a CLSID the client must now create an object of that class in order to make use of its services. It does so using two steps:

6. Obtain the "class factory" for the CLSID.
7. Ask the class factory to instantiate an object of the class, returning an interface pointer to the client.

After these steps, illustrated in Figure 5-1, the client is free to do whatever it wishes with the object through whatever interfaces the object supports. In fact, *everything* done with the object is accomplished through calls to interface member functions—APIs that seems to affect objects through other means are merely wrappers to common sequences of interface calls.

Before examining each of these steps, let's take a look at what a class factory is in the first place.

1.2.1The Class Factory Object: IClassFactory Interface

The class factory is another object itself that exists to *manufacture* objects (hence the name "factory") of a specific class (hence the qualifier "class").[1] A class factory object is implemented by a server module, either a DLL or EXE, and supports the IClassFactory interface described below. For the purposes of COM Clients, the IClassFactory interface is and interface on an object used by a client. For information on implementation, see Chapter 6, "COM Servers."

[1] Note that IClassFactory would be more appropriately be named IObjectFactoy since using it one creates objects, not classes. But IClassFactory remains for historical reasons.

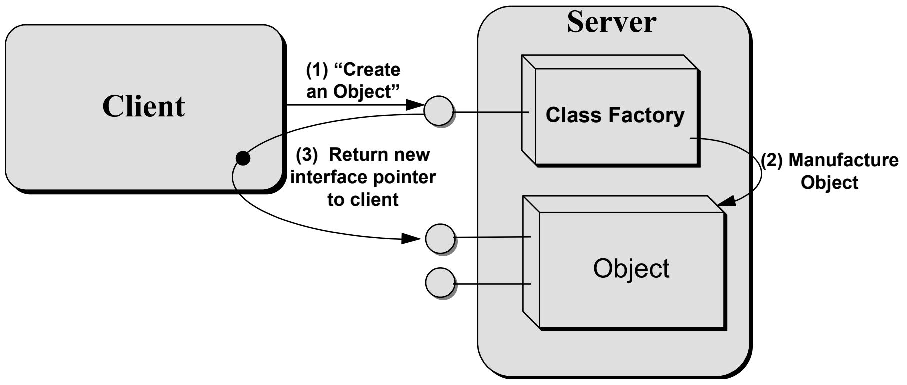

Figure 5-1 A client asks a class factory in the server to create an object.

The IClassFactory interface is implemented by COM servers on a "class factory" object for the purpose of creating new objects of a particular class. The interface also provides for a COM client to keep the server in memory even when it is not servicing any object. A class factory has a one-to-one correspondence with a CLSID (although actual implementations can be made generic to service multiple classes if the COM server so chooses).

```
[
   object,
   uuid(00000001-0000-0000-C000-000000000046), // IID_IClassFactory
   pointer_default(unique)
]
interface IClassFactory : IUnknown
{
      HRESULT CreateInstance([in] IUnknown * pUnkOuter, [in] REFIID iid, [out] void * ppv);
      HRESULT LockServer([in]BOOL fLock);
}
```

IClassFactory::CreateInstance

HRESULT IClassFactory::CreateInstance(pUnkOuter, iid, ppvObject)

Create an uninitialized instance, that is, object, of the class associated with the class factory, returning an interface pointer of type iid on the object to the caller in the out-parameter ppvObject.

If the object is being created as part of an aggregate—that is, the client of the object in this case is also an object server itself—then pUnkOuter contains the IUnknown pointer to the "outer unknown." See "Object Reusability" in Chapter 6 for more information. Class implementations need to be consciously designed to be aggregatable and accordingly not all classes are so designed.

Argument	Type	Description
pUnkOuter	IUnknown *	The controlling unknown of the aggregate object if this object is being created as part of an aggregate. If NULL, then the object is not being aggregated, which is the case when the object is being created from a pure client. If non-NULL and the class does not support aggregation, then the function returns CLASS_E_NOAGGREGATION.
iid	REFIID	The identifier of the first interface desired by the caller through which it will communicate with the object; usually the "initialization interface."
ppv	void **	The place in which the first interface pointer is to be returned.

Return Value	Meaning
S_OK	Success. A new instance was created.
E_NOAGGREGATION	Use of aggregation was requested, but this class does not support it.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

IClassFactory::LockServer

HRESULT IClassFactory::LockServer(fLock)

This function can be called by a client to keep a server in memory even when it is servicing no objects. Normally a server will unload itself (an EXE server) or allow the COM library to unload it (a DLL server) when the server has no objects left to serve. If the client so desires, it can lock the server in memory to prevent it from being loaded and unloaded multiple times, which can improve performance of object instantiations. Most clients have no need to call this function. It is present primarily for the benefit of sophisticated clients with special performance needs from certain classes.

It is an error to call LockServer(TRUE) and then call Release without first releasing the lock with LockServer(FALSE). Whoever locks the server is responsible for unlocking it, and once the class factory is released, there is no mechanism by which the caller can be guaranteed to later connect to the same class factory. All calls to IClassFactory::LockServer must be counted, not only the last one. Calls will be balanced; that is, for every LockServer(TRUE) call, there will be a LockServer(FALSE) call. If the lock count and the class object reference count are both zero, the class object can be freed.

For more information on the use of LockServer, see the "Server Management" section below. For more information on implementing this function, see Chapter 6 under "The Class Factory: Implementation and Exposure."

Argument	Type	Description
fLock	BOOL	True if a lock is being added to the class factory; false if one is being removed.

Return Value	Meaning
S_OK	Success.
E_UNEXPECTED	An unknown error occurred.

1.3Obtaining the Class Factory Object for a CLSID

Now that we understand what a class factory is and what functions it performs through the IClassFactory interface we can examine how a client obtains the class factory. This depends only slightly on whether the object in question is in-process, local, or remote. For the most part, all cases are handled through the same implementation locator service in the COM library and the same API functions. The implications are greater for servers as shown in Chapter 6.

For all objects on the same machine as the client, including object handlers, the client generates a call to the COM Library function CoGetClassObject. This function, described below, does whatever is necessary to obtain a class factory object for the given CLSID and return one of that class factory's interface pointers to the client. After that the client may calls IClassFactory::CreateInstance to instantiate objects of the class.

We say here that the client must *generate* a call to CoGetClassObject because it is not always necessary to call this function directly. When a client only wants to create a single object of a given class there is no need to go through the process of calling CoGetClassObject, IClassFactory::CreateInstance, and IClassFactory::Release. Instead it can use API function CoCreateInstance described below which conveniently wraps these three more fundamental steps into one function.

CoGetClassObject

HRESULT CoGetClassObject(clsid, grfContext, pServerInfo, iid, ppv)

Locate and connect to the class factory object associated with the class identifier clsid. If necessary, the COM Library dynamically loads executable code in order to accomplish this. The interface by which the caller wishes to talk to the class factory object is indicated by iid; this is usually IID_IClassFactory but can, of course, be any other object-creation interface.[2] The class factory's interface is returned in ppv with one reference count on it on behalf of the caller, that is, the caller is responsible for calling Release after it has finished using the class factory object.

[2] For example, the remoting architechture described in Chapter 7 uses a different type of "factory" interface.

Different pieces of code can be associated with one CLSID for use in different execution contexts such as in-process, local, or object handler. The context in which the caller is interested is indicated by the grfContext parameter, a group of flags taken from the enumeration CLSCTX:

```
typedef enum tagCLSCTX {
    CLSCTX_INPROC_SERVER        = 1,
    CLSCTX_INPROC_HANDLER       = 2,
    CLSCTX_LOCAL_SERVER         = 4,
    CLSCTX_REMOTE_SERVER        = 16.
    } CLSCTX;
```

The several contexts are tried in the sequence in which they are listed here. Multiple values may be combined (using bitwise OR) indicating that multiple contexts are acceptable to the caller:

```
#define CLSCTX_INPROC     (CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER)
#define CLSCTX_SERVER     (CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER)
#define CLSCTX_ALL        (CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER | CLSCTX_LOCAL_SERVER |
        CLSCTX_REMOTE_SERVER)
```

These context values have the following meanings which apply to all remote servers as well:

Value	Action Taken by the COM Library
CLSCTX_INPROC_SERVER	Load the in-process code (DLL) which creates and completely manages the objects of this class. If the DLL is on a remote machine, invoke a surrogate server as well to load the DLL.
CLSCTX_INPROC_HANDLER	Load the in-process code (DLL) which implements client-side structures of this class when instances of it are accessed remotely. An object handler generally implements object functionality which can only be implemented from an in-process module, relying on a local server for the remainder of the implementation.[3]
CLSCTX_LOCAL_SERVER	Launch the separate-process code (EXE) which creates and manages the objects of this class.[4]
CLSCTX_REMOTE_SERVER	Launch the separate-process code (EXE) on another machine which creates and manages objects of this class.

The COM Library should attempt to load in-process servers first, then in-process handlers, then local servers, then remote servers. This order helps to minimize the frequency with which the library has to launch separate server applications which is generally a much more time-consuming operation than loading a DLL, especially across the network.

When specifying CLSCTX_REMOTE_SERVER, the caller may pass a COMSERVERINFO structure to indicate the machine on which to run the server module, which is defined as follows:

```
typedef struct tagCOMSERVERINFO {
    OLECHAR      *szRemoteSCMBindingHandle;
    } COMSERVERINFO;[5]
```

The COM Library implementation of this CoGetClassObject relies on the system registry to map the CLSID to the server module to load or launch, but this process is opaque to the client application. If, however, COM cannot make any association then the function fails with the code REGDB_E_CLASSNOTREG. If this function launches a server application it must wait until that server registers its class factory or until a time-out occurs (duration determined by COM, something on the order of a minute of processing time). See the CoRegisterClassObject function in Chapter 6 under “Exposing the Class Factory from Local Servers.”

[3] For example, in OLE 2, built on top of COM, there is an interface called IViewObject through which a client can ask an object to draw its graphical presentation directly to a Windows device context (HDC) through IViewObject::Draw. However, an HDC cannot be shared between processes, so this interface can only be implemented inside as part of an in-process object. When an object server wishes to provide optimized graphical output but does not wish to completely implement the object in-process, it can use a lightweight object handler to implement just the drawing functionality where it must reside, relying on the local server for the full object implementation.

[4] In some cases the object server may already be running and may allow its class factory to be used multiple times in which case the COM Library simply establishes another connection to the existing class factory in that server, eliminating the need to launch another instance of the server applications entirely. While this can improve performance significantly, it is the option of the server to decide if its class factory is single- or multiple-use. See the function CoRegisterClassObject in Chapter 6 for more information.

[5] This abstraction is still under design.

The arguments to this function are as follows:

Argument	Type	Description
clsid	REFCLSID	The class of the class factory to obtain.
grfContext	DWORD	The context in which the executable code is to run.
pServerInfo	COMSERVERINFO*	Identifies the machine on which to activate the executable code. Must be NULL when grfContext does not contain CLSCTX_REMOTE_SERVER. When NULL and grfContext contains CLSCTX_REMOTE_SERVER, COM uses the default machine location for this class.
iid	REFIID	The interface on the class factory object desired by the caller.
ppv	void **	The place in which to put the requested interface.

Return Value	Meaning
S_OK	Success.
REGDB_E_CLASSNOTREG	An implementation of the requested class could not be located.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

The following code fragment demonstrates how a client would call CoGetClassObject and create an in-process instance of the TextRender object with CLSID_TextRender using the class factory to request an IUnknown pointer for the object. In this example the client is explicitly limiting COM to use only in-process servers:

```
IClassFactory *pCF;
IUnknown *      pUnkObj;
HRESULT             hr;

hr=CoGetClassObject(CLSID_TextRender, CLSCTX_INPROC_SERVER, NULL, IID_IClassFactory, (void *)pCF);
if (FAILED(hr))
      //Could not obtain class factory, creation fails completely.

/*
 * Create the object. If this call succeeds the pUnkObj will
 * be valid and have a reference count on it on behalf of the caller
 * which the caller must Release.
 */
hr=pCF->CreateInstance(NULL, IID_IUnknown, (void *)pUnkObj);

//Caller must call Release regardless of CreateInstance result
pCF->Release();

if (FAILED(hr))
      //Object creation failed: interface may not be supported

/*
 * Now use the object in whatever capacity the caller desires.
 * The first step will be initialization.
 */

//Release the object when finished with it.
pUnkObj->Release();
```

Since the process of calling CoGetClassObject, IClassFactory::CreateInstance, and IClassFactory::Release is so common in practice, the COM Library provides a wrapper API function for this sequence called CoCreateInstance. This allows the client to avoid the whole issue of class factory objects entirely. However, CoCreateInstance only creates one object at a time; if the client wants to create multiple objects of the same class at once, it is more efficient to obtain the class factory directly and call IClassFactory::CreateInstance multiple times, avoiding excess calls to CoGetClassObject and IClassFactory::Release.

CoCreateInstance

HRESULT CoCreateInstance(clsid, pUnkOuter, grfContext, iid, ppvObj)

Create an uninitialized instance of the class clsid, asking for interface iid using the execution contexts given in grfContext. If the object is being used as part of an aggregation then pUnkOuter contains a pointer to the controlling unknown. These parameters behave as those of the same name in CoGetClassObject (clsid) and IClassFactory::CreateInstance (pUnkOuter, grfContext, iid, ppv),

CoCreateInstance is simply a wrapper function for CoGetClassObject and IClassFactory that is implemented (conceptually) as follows:

```
HRESULT CoCreateInstance(REFCLSID clsid, IUnknown * pUnkOuter,
    DWORD grfContext, REFIID iid, void * ppvObj)
    {
    IClassFactory * pCF;
    HRESULT       hr;

    hr=CoGetClassObject(clsid, grfContext, NULL, IID_IClassFactory, (void *)pCF);

    if (FAILED(hr))
        return hr;

    hr=pCF->CreateInstance(pUnkOuter, iid, (void *)ppv);
    pCF->Release();

    /*
     * If CreateInstance fails, ppv will be set to NULL. Otherwise
     * ppv has the interface pointer and hr contains NOERROR.
     */
    return hr;
    }
```

Argument	Type	Description
clsid	REFCLSID	The class of which an instance is desired
pUnkOuter	IUnknown*	The controlling unknown, if any.
grfContext	DWORD	The CLSCTX to be used.
iid	REFIID	The initialization interface desired
ppv	void**	The place at which to return the desired interface.

Return Value	Meaning
S_OK	Success.
Any error that can be returned from CoGetClassObject or IClassFactory::CreateInstance	Semantics as in those functions.
E_UNEXPECTED	An unknown error occurred.

CoCreateInstanceEx

HRESULT CoCreateInstanceEx(clsid, pUnkOuter, grfContext, pServerInfo, dwCount, rgMultiQI)

Create an uninitialized instance of the class clsid on a specific machine, asking for a set of interface iids in pResult using the execution contexts given in grfContext. If the object is being used as part of an aggregation then pUnkOuter contains a pointer to the controlling unknown.

To help optimize round-trips to a remote machine during instantiation, this API allow the client to specify a set of interfaces to return on the object via the rgMultiQI array of MULTI_QI structures, defined as follows:

```
typedef struct tagMULTI_QI {
    REFIID      riid;       // interface to return
    void*       pvObj;      // location to return interface pointer
    HRESULT     hr;         // location to return result of QueryInterface for riid
    } MULTI_QI;
```

The semantics of using this API and passing a MULTI_QI array are identical to the following sequence of operations, but incur less overhead for the client, the server, and the network:

```
IClassFactory   *pCF;
IUnknown        *punk;
COMSERVERINFO csi;

CoGetClassObject(clsid, CLSCTX_SERVER, &csi, IID_IClassFactory, (void**)&pCF);
pCF->CreateInstance(NULL, IID_IUnknown, (void**)&punk);
for (DWORD i=0; i<dwCount; i++)
        rgMultiQI[I].hr = punk->QueryInterface(rgMultiQI[i].riid, &rgMultiQI[i].pvObj);
punk->Release();
```

Argument	Type	Description
clsid	REFCLSID	The class of which an instance is desired
pUnkOuter	IUnknown*	The controlling unknown, if any.
grfContext	DWORD	The CLSCTX to be used.
pServerInfo	COMSERVERINFO*	Identifies the machine on which to activate the executable code. Must be NULL when grfContext does not contain CLSCTX_REMOTE_SERVER. When NULL and grfContext contains CLSCTX_REMOTE_SERVER, COM uses the default machine location for this class.
dwCount	DWORD	The number of MULTI_QI structures in the rgMultiQI array.
rgMultiQI	MULTI_QI*	An array of MULTI_QI structures. On input, each element should be cleared and the riid member set to an IID being requested. On output, one or more of the interfaces may be retrieved, and individual pvObj members will be non-NULL.

Return Value	Meaning
S_OK	Success.
CO_S_NOTALLINTERFACES	Not all of dwCount interfaces requested in the MULTI_QI array were successfully retrieved. Examine individual pvObj members of MULTI_QI to determine exactly which interfaces were returned.
Any error that can be returned from CoGetClassObject or IClassFactory::CreateInstance	Semantics as in those functions.
E_UNEXPECTED	An unknown error occurred.

1.4Initializing the Object

After the client has successfully created an object of a given class it must initialize that object. By definition, any new object created using IClassFactory::CreateInstance (or variant or wrapper thereof) is uninitialized. Initialization generally happens through a single call to a member function of the "initialization interface." This interface is usually the one requested by the client in its call to create the object, but this is not required. Before an object is initialized, the only calls that are guaranteed to work on the object (besides the initializing functions themselves) are the IUnknown functions (of any interface) unless otherwise explicitly specified in the definition of an interface. In addition, QueryInterface is only guaranteed to work for IUnknown and any initialization interface, but not guaranteed for a non-initialization interface.

Some objects will not require initialization before they are function through all of their interfaces. Those that do require initialization will define, either explicitly through documentation of the object or implicitly through the scenarios in which the object is used, which member of which interface can be used for initialization.

For example, objects that can serialize their persistent data to a file will implement the IPersistFile interface (see "Persistent Storage Interfaces for Objects" in Chapter 8). The function IPersistFile::Load, which instructs the object to load its data from a file, is the initialization function and IPersistFile is the initialization interface. Other examples are objects that can serialize to storages or streams, where the objects implement the initialization interfaces IPersistStorage or IPersistStream, respectively (again, see Chapter8). The Load functions in these interfaces are initialization functions as is IPersistStorage::InitNew, which initializes a new object with storage instead of loading a previously saved version.

1.5Managing the Object

Once an object is initialized, it is entirely up to the client to determine what it intends to do with that object. It is often the case that the initializing interface is not the “working” interface through which the client will primarily use the object. The creation sequence only nets the client a single interface pointer that has a limited scope of functionality. If the client wishes to perform an operation outside that scope, it must call the known interface’s QueryInterface function to ask for another interface on the same object.

For example, say a client has created and initialized an object but now wishes to obtain a graphical presentation, say a bitmap, from that object by calling IDataObject::GetData (see Chapter 10 for details on this function). The client must call QueryInterface to obtain an IDataObject pointer before calling the function.

It is important to note that *all operations on that object will occur through calls to the member functions of the object’s various interfaces*. Any additional API functions that the client might call to affect the object itself are usually wrapper functions of common sequences of interface function calls. There simply is no other way to affect the object other than through it’s interfaces.

Because a client must ask for an interface before it can possibly ask the object to perform the actions defined in the interface, the client cannot ask the object to perform an action the object does not support. This is a primary strength of the QueryInterface function as described in the early chapters of this document. Calling QueryInterface for access to an object’s functionality is not problematic nor inconvenient because the client usually makes the call specifically at the point where the client wants to perform some action on the object. That is, clients generally do not call QueryInterface for all possible interfaces after the object is created so as to have all the pointers on hand—instead, the client calls QueryInterface before attempting to perform some action with the object.

In practice this means that the client must be prepared for the failure of a call to QueryInterface. Instead of being a complete pain to implementation, such preparation defines a mechanism through which the client can make dynamic choices based on the functionality of the object itself on an object-by-object basis.

For example, consider a client application that has created a number of objects and it now wants to save the application’s state, which includes saving the state of each object. Let’s say the client is using structured storage for its native file representation, so its first choice will be to assign an individual storage element in that file for each object. Each object can then store structured information itself and it indicates its ability to do by implementing the IPersistStorage interface. However, some object may not know how to write to a storage but know how to write to a stream and indicate the capability by implementing IPersistStream. Yet others may only know how to write information to a file themselves and thus implement IPersistFile. Finally, some objects may not know how to serialize themselves at all, but can provide a binary memory copy of the their native data through IDataObject.

In this case the client’s strategy will be as follows: if an object supports IPersistStorage, then give it an IStorage instance and ask it to save its data into it by calling IPersistStorage::Save. If that object does not provide such support, check if it supports IPersistStream, and if so, create a client-controlled stream for it (in perhaps a separate client-controlled storage element) and pass that IStream pointer to the object through IPersistStream::Save. If the object does not support streams, then check for IPersistFile. If the object supports serialization to a file, then have the object write its data into a temporary file by calling IPersistFile::Save, then make a binary copy of that file in a client-controlled stream element within a client-controlled storage element. If all else fails, attempt to retrieve the object’s binary data from IDataObject::GetData using the first format the object supports, and write that binary data into a client-controlled stream in a client-controlled storage.

Code for such a strategy would be structured something like the following pseudo-code for a “save object” function in the client:

```
BOOL SaveObject(IUnknown * pUnkObj)
    {
    pUnkObj->QueryInterface(IID_IPersistStorage)

    if (success)
        {
        create a storage element for the object
        call IPersistStorage::Save
        call IPersistStorage::Release
        return TRUE
        }

    //All other cases use a client-controlled stream
    create a stream element for the object in some storage

    //IPersistStorage not supported, try IPersistStream
    pUnkObj->QueryInterface(IID_IPersistStream)

    if (success)
        {
        call IPersistStream::Save
        call IPersistStream::Release
        return TRUE
        }

    //IPersistStream not supported, try IPersistFile
    pUnkObj->QueryInterface(IID_IPersistFile)

    if (success)
        {
        //Save to a temp file
        call IPersistFile::Save("objdata.tmp");
        call IPersistFile::Release
        read data from temp file
        write data to the stream
        return TRUE
        }

    //All else failed, try IDataObject
    pUnkObj->QueryInterface(IID_IDataObject)

    if (success)
        {
        call IDataObject::EnumFormatEtc
        call IEnumFORMATETC to get the first format (assume it's native)
        call IEnumFORMATETC::Release

        call IDataObject::GetData for the format, asking for global memory
        call IDataObject::Release

        Lock global memory and write to stream
        Free global memory
        return TRUE
        }

    //Everything failed, so give up
    destroy stream we created:  not using it.
    return FALSE
    }
```

In this example the client is prepared for many different types of objects and how they might provide persistent information (and using IDataObject::GetData here is stretching the concept somewhat, but shows that the client has many choices). Based on the results of QueryInterface the client decides at run-time how to save each individual object.

Reloading these objects would be a similar procedure, but the client would know, from the structure of its storage and other information it saved about the objects itself, which method to use to reload the object from the storage. The client wants to insure that it uses the same method to load the object that it did for saving it originally, that is, use the same interface instead of querying for the best one. The reason is that while the data was passively stored on disk, the object that wrote that data might have been updated such that where it once only supported IPersistStream, for example, it now supports IPersistStorage. In that case the client should ask it to load the data using IPersistStream::Load.

However, when the client goes to save the object again, it will now successfully find that the object supports IPersistStorage and can now have the object save into a storage element instead. (The container would also insure that the old client-controlled stream was deleted as it is no longer in use for that object.) This demonstrates how an object can be updated and new interfaces supported without *any* recompilation on the part of existing clients while at the same time *suddenly working with clients on a higher level of integration than before*. In order to remain compatible the object must insure that it supports the older interfaces (such as IPersistStream) but is free to add new contracts—new interfaces such as IPersistStorage—as it wants to provide new functionality.

The point of this example, which is also true for clients that use any other interfaces an object might support in other scenarios, is that the client is empowered to make dynamic decisions on a per-object basis through the QueryInterface function. Containers programmed to be dynamic as such allow object to improve independently while insuring that the container will work as good—and generally better—as it always has with any given object. All of this is due to the powerful and important QueryInterface mechanism that for all intents and purposes is the single most important aspect of true system component software.

1.6Releasing the Object

The final operation required in a COM client when dealing with an object from some other server is to free that object when the client no longer needs it. This is achieved by calling the Release member function of all interfaces obtained during the course of using the object.

Recall that a function that creates or synthesizes a new interface pointer is responsible for calling AddRef through that pointer before returning it to the caller of the function. This applies to the IClassFactory::CreateInstance function as well as CoCreateInstance (and for that matter, CoGetClassObject, too, which is why you must call IClassFactory::Release after creating the object). Therefore, as far as the client is concerned, the object will have a reference count of one after creation. The object may, in fact, have a higher reference count if it is also being used from other clients as well, but each client is only responsible and cognizant of the reference counts added on its behalf.

The other primary function that creates new interface pointers is QueryInterface. Every call the client makes to QueryInterface to obtain another interface pointer will internally generate another call to AddRef in that object, incrementing the reference count. Therefore, in addition to calling Release through the interface pointer obtained in the creation sequence, the client must also call Release through any interface pointer obtained from QueryInterface (this is illustrated in the pseudo-code of the previous section).

The bottom line is that the client is responsible for matching any operation that generates a call to AddRef through a given interface pointer with a call to Release through that same interface pointer. It is not necessary to call Release in the opposite order of calls to AddRef; it is just necessary to match the pairs. Failure to do so will cause memory leaks as objects are not freed and servers are not allowed to shut down properly. This is no different that forgetting to free memory obtained through malloc.

Finally, although the client matches its calls to AddRef and Release, the actual object may still continue to run and the server may continue to execute as well without any objects in service. The object will continue if other clients are using that same object and thus have reference counts on it. Only when all clients have released their references will that object free itself. The server will, of course, continue to execute as long as there is an object to serve, but the client does have some power over keeping a server running even without objects. That is the purpose of Server Management functions in COM.

1.7Server Management

As mentioned in previous sections, a client has the ability to manage servers on the server level to keep them running even when they are not serving any objects. The client's primary mechanism for this is the IClassFactory::LockServer function described above. By calling this function with the TRUE parameter, the client places a 'lock' on the server. As long as the server either has objects created *or* has one or more locks on it, the server will continue to execute. When the server detects a zero object and zero lock condition, it can unload itself (which differs between DLL and EXE servers, as described in Chapter 7).

A client can place more than one lock on a server by calling IClassFactory::LockServer(TRUE) more than once. Each call to LockServer(TRUE) must be matched with a call to LockServer(FALSE)—the server maintains a lock count for the server as it maintains a reference count for its served objects. But while AddRef and Release affect objects, LockServer affects the server itself.

LockServer affects all servers—in-process, local, and remote—identically. The client does have some additional control over in-process objects as it normally would for other DLLs through the functions CoLoadLibrary, CoFreeUnusedLibraries, and CoFreeAllLibraries, as described below. Normally only CoFreeUnusedLibraries is called from a client whereas the others are generally used inside the COM Library to implement other API functions. In addition, the COM Library supplies one additional function that has meaning in this context, CoIsHandlerConnected, that tells the container if an object handler is currently working in association with a local server as described in its entry below.

CoFreeUnusedLibraries

void CoFreeUnusedLibraries(void)

This function and unloads any DLLs that have been loaded as a result of COM object creation calls but which are no longer in use. Client applications can call this function periodically to free up resources.

CoIsHandlerConnected

BOOL CoIsHandlerConnected(pUnk)

Determines if the specified handler is connected to its corresponding object in a running local server. The result of this function might be used in a client application to determine if certain operations might result in launching a server application allowing the client to make performance decisions.

Argument	**Type**	**Description**
pUnk	IUnknown *	Specifies the object in question.
return value	BOOL	True if a handler is connected to a running server with the full object implementation, FALSE if the handler is not connected.