1COM Servers

As described in earlier chapters, a COM Server is some module of code, a DLL or an EXE, that implements one or more object classes (each with their own CLSID). A COM server structures the object implementations such that COM clients can create an use objects from the server using the CLSID to identify the object through the processes described in Chapter 5.

In addition, COM servers themselves may be clients of other objects, usually when the server is using those other objects to help implement part of its own objects. This chapter will cover the various methods of using an object as part of another through the mechanisms of containment and aggregation.

Another feature that servers might support is the ability to emulate a different server of a different CLSID. The COM Library provides a few API functions to support this capability that are covered at the end of this chapter.

If the server is an application, that is, an executable program, then it must follow all the requirements for a COM Application as detailed in Chapter 4. If the server is a DLL, that is, an in-process server or an object handler, it must at least verify the library version and may, if desired, insure that the COM Library is initialized. That aside, all servers generally perform the following operations in order to expose their object implementations:

1. Allocate a class identifier—a CLSID—for each supported class and provide the system with a mapping between the CLSID and the server module.
2. Implement a class factory object with the IClassFactory interface for each supported CLSID.
3. Expose the class factory such that the COM Library can locate it after loading (DLL) or launching (EXE) the server.
4. Provide for unloading the server when there are no objects being served and no locks on the server (IClassFactory::LockServer).

Of course, there must be some object to serve, so the first section of this chapter discusses the basic structure of an object and some considerations for design. The sections that follow then cover the functions involved in each of these steps for the different styles of servers—DLL and EXE—which apply regardless of whether the server is running on a remote machine. Also included is a discussion of object handlers (special-case in-process objects) before the discussion of aggregation. Note that no new interfaces are introduced in this chapter as the fundamental ones, IUnknown and IClassFactory, have already been covered.

As far as the server is concerned, the COM Library exists to drive the server's class factory to create objects and to handle remote method calls from clients in other processes or on other machines and to marshal the object's return values back to the client. Whereas client applications are unaware of the object's execution context once the object is created, the server is, of course, always aware of that context. An in-process object is always loaded into the client's process space. A local or remote object always runs in a process other than the client, or on a different machine. However, the actual object itself can be written such that it does not need to care about the execution context, leaving the specifics to the structure of the server module instead. This chapter will cover one such strategy.

Finally, recall from the beginning of Chapter 5 that a client always makes a call into some in-process object whenever it calls any interface member function. If the actual object in the server is local or remote, that object is merely a proxy that generates the appropriate remote method call to the true object. This does not mean a server has to understand RPC, however, as the server always sees these calls as direct calls from a piece of code in the server process. The mechanism that achieves this, described in Chapter 7, "Communicating via Interfaces: Remoting," is that the RPC call is picked up in the server process by an "stub" object which translate the RPC information into the direct call to the server's object. From the server's point of view, the client called it directly.

1.1Identifying and Registering an Object Class

A major strength of COM is the use of globally unique identifiers to essentially name each object class that exists, not only on the local machine but universally across all machines and all platforms. The algorithm that guarantees this is encompassed in the COM Library function CoCreateGuid as described in Chapter 3. An object implementor must obtain a GUID to assign to the object server as its CLSID for each implemented class.

1.1.1System Registry of Classes for the Local Machine

A CLSID to identify an object implementation is not very useful unless clients have a way of finding the CLSID. From Chapter 5 we know that there are a number of ways a client may come to know a CLSID. First of all, that client may be compiled to specifically depend on a specific CLSID, in which case it obtained the server's header files with the DEFINE_GUID macros present. But for the most part, clients will want to obtain CLSIDs at run-time, especially when that client displays a list of available objects to and end-user and creates an object of the selected type at the user's request. So there must be a way to dynamically locate and load CLSIDs for accessible objects.

Furthermore, there has to be some system-wide method for the COM Library to associate a given CLSID, regardless of how the client obtained it, to the server code that implements that class. In other words, the COM Library requires some persistent store of CLSID-to-server mappings that it uses to implement its locator services. It is up to the COM Library implementor, not the implementor of clients or servers, to define the store and how server applications would register their CLSIDs and server module names in that store.

The store must distinguish between in-process, local, and remote objects as well as object handlers in addition to any environment-specific differences. The COM implementation on Microsoft Windows uses the Windows system registry (also called the registration database, or RegDB for short) as a store for such information. In that registry there is a root key called "CLSID" (spelled out in those letters) under which servers are responsible to create entries that point to their modules. Usually these entries are created at installation time by the application's setup code, but can be done at run-time if desired.

When a server is installed under Windows, the installation program will create a subkey under "CLSID" for each class the server supports, using the standard string representation of the CLSID as the key name (including the curly braces).[1] So the first key that the TextRender object would create appears as follows (CLSID is the root key the indentation of the object class implies a sub-key relationship with the one above it):

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
```

Depending on the type of same-machine server that handles this CLSID there will be one or more subkeys created underneath the ASCII CLSID string:

Server Flavor	Subkey Name	Value
In-Process	InprocServer32	Pathname of the server DLL
Local	LocalServer32	Pathname of the server EXE
Object Handler	InprocHandler32	Pathname to the object handler DLL.

So, for example, if the TextRender object was implemented in a TEXTREND.DLL, its entries would appear as:

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
        InprocServer32 = c:\objects\textrend.dll
```

[1] Under Microsoft Windows, this key is created using the standard Windows API for registry manipulation. Other COM implementations may include their own functions as necessary, as long as it's consistent on a given platform. Such functions are not part of this specification

If it were implemented in an application, TEXTREND.EXE, and worked with an object handler in TEXTHAND.DLL, the entries would appear as:

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
        InprocHandler32 = c:\handlers\texthand.dll
        LocalServer32 = c:\objects\textrend.exe
```

Over time, the registry will become populated with many CLSIDs and many such entries.

1.1.2Remote Objects: AtBits Key

As described in the last section, a prerequisite to server implementation is generating a CLSID for that server. This CLSID is registered in the system registry and referenced in the server code. The full path name of the server DLL or EXE is registered in association with the CLSID.

The remote server can actually run either on the machine where the server code is stored or on the same machine as its connected client (assuming the class is registered on the remote machine and there is a compatible binary image available). Servers that use the default security provided with the system must run where its client is running. To indicate the mode of operation, the Microsoft Windows implementation of COM includes the subkey "AtBits" that is registered along with the server's CLSID. To register a server to run where the persistent state of the object is stored, set AtBits to "Y." To register the server to run where the client is running, either set it to "N" or leave the attribute out altogether. The default is to run the server where the client is running. The registration example below shows how the TextRender object would allow itself to be activated remotely.

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
        LocalServer = c:\objects\textrend.exe
        AtBits = Y
```

1.1.3Self-Registering Servers

COM servers which are installed as part of an application setup program are usually registered by the setup program. However, to facilitate the registration of smaller grained servers, the notion of a self-registering server is introduced.

Self-Registering DLL's

In-process COM servers (DLL's on the Windows and Macintosh platforms) support self-registration through several DLL entry points with well-known names. The DLL entry points for registering and unregistering a server are defined as follows:

```
HRESULT DllRegisterServer(void);
HRESULT DllUnregisterServer(void);
```

Both of these entry points are required for a DLL to be self-registering. The implementation of the DllRegisterServer entry point adds or updates registry information for all the classes implemented by the DLL. The DllUnregisterServer entry point removes its information from the registry.

Self-Registering EXE's

There isn't an easy way for EXE's to publish entry points with well-known names, so a direct translation of DllRegisterServer isn't possible. Instead, EXE's support self-registration using special command line flags. EXE's that support self-registration must mark their resource fork in the same way as DLL's, so that the EXE's support for the command line flags is detectable. Launching an EXE marked as self-registering with the /REGSERVER command line argument should cause it to do whatever OLE installation is necessary and then exit. The /UNREGSERVER argument is the equivalent to DllUnregisterServer. The /REGSERVER and /UNREGSERVER strings should be treated case-insensitively, and that the character '-' can be substituted for '/'.

Other than guaranteeing that it has the correct entry point or implements the correct command line argument, an application that indicates it is self-registering must build its registration logic so that it may be called any number of times on a given system even if it is already installed. Telling it to register itself more than once should not have any negative side effects. The same is true for unregistering.

On normal startup (without the /REGSERVER command line option) EXE's should call the registration code to make sure their registry information is current. EXE's will indicate the failure or success of the self-registration process through their return code by returning zero for success and non-zero for failure.

Identifying Self-Registering Servers

Applications need to check to see if a given server module is self-registering without actually loading the DLL or EXE for performance reasons and to avoid possible negative side-affects of code within the module being executed without the module first being registered. To accomplish this, the DLL or EXE must be tagged with a version resource that can be read without actually causing any code in the module to be executed. On Windows platforms, this involves using the version resource to hold a self-registration keyword. Since the VERSIONINFO section is fixed and cannot be easily extended, the following string is added to the "StringFileInfo", with an empty key value:

```
VALUE "OLESelfRegister", ""
```

For example:

```
VS_VERSION_INFO    VERSIONINFO
 FILEVERSION      1,0,0,1
 PRODUCTVERSION   1,0,0,1
 FILEFLAGSMASK    VS_FFI_FILEFLAGSMASK
#ifdef _DEBUG
 FILEFLAGS       VS_FF_DEBUG|VS_FF_PRIVATEBUILD|VS_FF_PRERELEASE
#else
 FILEFLAGS       0 // final version
#endif
 FILEOS          VOS_DOS_WINDOWS16
 FILETYPE        VFT_APP
 FILESUBTYPE     0   // not used
BEGIN
 BLOCK "StringFileInfo"
 BEGIN
  BLOCK "040904E4" // Lang=US English, CharSet=Windows Multilingual
  BEGIN
  VALUE "CompanyName",     "\0"
  VALUE "FileDescription", "BUTTON OLE Control DLL\0"
  VALUE "FileVersion",     "1.0.001\0"
  VALUE "InternalName",    "BUTTON\0"
  VALUE "LegalCopyright",  "\0"
  VALUE "LegalTrademarks", "\0"
  VALUE "OriginalFilename","BUTTON.DLL\0"
  VALUE "ProductName",     "BUTTON\0"
  VALUE "ProductVersion",  "1.0.001\0"
  VALUE "OLESelfRegister", "" // New keyword
  END
 END
 BLOCK "VarFileInfo"
 BEGIN
  VALUE "Translation", 0x409, 1252
 END
END
```

To support self-registering servers, an application can add a "Browse" button to its object selection user interface, which pops up a standard File Open dialog. After the user chooses a DLL or EXE the application can check to see if it is marked for self-registration and, if so, call its DllRegisterServer entry point (or execute the EXE with the /REGSERVER command line switch). The DLL or EXE should register itself at this point.

1.2Implementing the Class Factory

The existence of a CLSID available to clients implies that there is a class factory that is capable of manufacturing objects of that class. The server, DLL or EXE, associated with the class in the registry is responsible to provide that class factory and expose it to the COM Library to make COM's creation mechanisms work for client. The specific mechanisms to expose the class factory is covered shortly, but first, let's examine how a class factory may be implemented[2].

1.2.1Defining the Class Factory Object

First of all, you need to define an object that implements the IClassFactory interface (or other factory-type interface if applicable). As you would define any other object, you can define a class factory. The following is an example class factory for our TextRender objects in C++:

```
class CTextRenderFactory : public IClassFactory
    {
    protected:
        ULONG           m_cRef;

    public:
        CTextRenderFactory(void);
        ~CTextRenderFactory(void);

        //IUnknown members
        HRESULT QueryInterface(REFIID, pLPVOID);
        ULONG AddRef(void);
        ULONG Release(void);

        //IClassFactory members
        HRESULT CreateInstance(IUnknown *, REFIID iid, void **ppv
        HRESULT LockServer(BOOL);
    };
```

Implementing the member functions of this object are fairly straightforward. AddRef and *Release* do their usual business, with Release calling *delete this* when the count is decremented to zero. Note that the zero-count event in *Release* has no effect other than to destroy the object—it does not cause the server to unload as that is the prerogative of LockServer. In any case, the QueryInterface implementation here will return pointers for IUnknown and IClassFactory.

IClassFactory::CreateInstance

The class factory-specific functions are really all that are interesting. CreateInstance in this example will create an instance of the CTextRender object and return an interface pointer to it as shown below. Note that if pUnkOuter is non-NULL, that is, another object is attempting to aggregate, this code will fail with CLASS_E_NOAGGREGATION (this limitation will be revisited when later when aggregation is discussed).

```
//A global variable that counts objects being served
ULONG    g_cObj=0;

HRESULT CTextRenderFactory::CreateInstance(IUnknown * pUnkOuter, REFIID iid, void ** ppv) {
    CTextRender *       pObj;
    HRESULT             hr;

    *ppv=NULL;
    hr=E_OUTOFMEMORY;
    if (NULL!=pUnkOuter)
        return CLASS_E_NOAGGREGATION;

    //Create the object passing function to notify on destruction.
    pObj=new CTextRender(pUnkOuter, ObjectDestroyed);
    if (NULL==pObj)
        return hr;

    [Usually some other object initialization done here]

    //Obtain the first interface pointer (which does an AddRef)
    hr=pObj->QueryInterface(iid, ppv);

    //Kill the object if initial creation or FInit failed.
    if (FAILED(hr))
        delete pObj;
```

2 Note that the example code given below illustrates one of many ways a class factory object can be implemented.

```
        else
            g_cObj++;

        return hr;
        }
```

There are two interesting points to this code, which is fairly standard for server implementations. First of all, note the call to the object's QueryInterface after creation. This accomplishes two things: first, since objects are generally constructed with a reference count of zero (common practice) then this QueryInterface call, if successful, has the effect of calling AddRef as well, making the object have a reference count of one. Second, it lets the object determine if it supports the interface requested in iid and if it does, it fills in ppv for us.

The second key point is that COM defines no standard mechanism for counting instantiated objects (there is no need for such a generic service), so this implementation example maintains a count of the objects in service using the global variable g_cObj. This count generally needs to be global so that other global functions can access it (see "Providing for Server Unloading" below). When CreateInstance successfully creates a new object it increments this count. When an object (not the class factory but the one the class factory creates) destroys itself in it's implementation of CTextRender::Release, it should decrement this count to match the increment in CreateInstance.

It is not necessary, however, for the object to have direct access to this variable, and there are techniques to avoid such access.. The example above passes a pointer to a function called ObjectDestroyed to the CTextRender constructor such that when the object destroys itself in it's Release it will call ObjectDestroyed to affect the server's object count:

```
    void ObjectDestroyed(void) {
        g_cObj--;
        [Initiate unloading if g_cObj is zero and there are no locks]
        return;
        }

    CTextRender::CTextRender(void (* pfnDestroy)(void)) {
        m_cRef=0;
        m_pfnDestroy=pfnDestroy;
        [Other initialization]
        return;
        }

    ULONG CTextRender::Release(void) {
        ULONG      cRefT;
        cRefT=--m_cRef;
        if (0L==m_cRef) {
            if (NULL!=m_pfnDestroy)
                (*m_pfnDestroy)();
            delete this;
            }
        return cRefT;
        }
```

The object might also be given a pointer to the class factory object itself (which the object will call AddRef through, of course) that accomplishes the same thing. Regardless of the design, the point is that the object can be designed so as to be unaware of the exact object counting mechanism, having instead some mechanism to notify the server as a whole about the destroy event. A standard mechanism for this is not part of COM.

You might have noticed that the ObjectDestroyed function above contained a note that if there are no objects and no locks on the server, then the server can initiate unloading. What really happens here depends on the type of server, DLL or EXE, and will be covered under "Providing for Server Unloading."

IClassFactory::LockServer

The other interesting member function of a class factory is LockServer. Here the server increments of decrements a lock count depending on the fLock parameter. If the last lock is removed and there are no objects in server, the server initiates unloading which again, is specific to the type of server and a topic for a later section. In any case, COM does not define a standard method for tracking the lock count. Since other code outside of the class factory may need access to the lock count, a global variable works well:

```
//Global server lock count.
ULONG     g_cLock=0;
```

The implementation of LockServer is correspondingly simple:

```
HRESULT CTextRenderFactory::LockServer(BOOL fLock)
    {
    if (fLock)
        g_cLock++;
    else
        {
        g_cLock--;
        [Initiate unloading if there are no objects and no locks]
        }

    return NOERROR;
    }
```

It is perfectly reasonable to double the use of g_cObj for counting locks as well as objects. You might want to keep them separate for debugging purposes.

1.3Exposing the Class Factory

With a class factory implementation the server must now expose it such that the COM Library can locate the class factory from within CoGetClassObject after it has loaded the DLL[3] server or launched the EXE server. The exact method of exposing the class factory differs for each server type. The following sections cover each type in detail which apply to DLLs and EXEs running on the local or remote machine in relation to the client. There are also some considerations for DLL servers running remotely under a surrogate server that are covered in this section.

1.3.1Exposing the Class Factory from DLL Servers

To expose its class factory, an in-process server only needs to export[4] a function explicitly named DllGetClassObject. The COM Library will attempt to locate this function in the DLL's exports[5] and call it from within CoGetClassObject when the client has specified CLSCTX_INPROC_SERVER. Note that a DLL server can in addition expose a class factory at a later time using the function CoRegisterClassObject discussed for EXE servers below. This would only be used after the DLL was already loaded for some other reason.

DllGetClassObject

HRESULT DllGetClassObject(clsid, iid, ppv)

This is not a function in the COM Library itself; rather, it is a function that is exported from DLL servers.

In the case that a call to the COM API function CoGetClassObject results in the class object having to be loaded from a DLL, CoGetClassObject uses the DllGetClassObject that must be exported from the DLL in order to actually retrieve the class.

Argument	Type	Description
clsid	REFCLSID	The class of the class factory being requested.
iid	REFIID	The interface with which the caller wants to talk to the class factory. Most often this is IID_IClassFactory but is not restricted to it.
ppv	void **	The place in which to put the interface pointer.

Return Value	Meaning
S_OK	Success.

[3] Again, the term DLL is used generically to describe any shared library as supported by a given COM platform.

[4] Under Microsoft Windows this means listing the function in the EXPORTS section of a module definitions file or using the __declspec(dllexport) keyword at compile-time. Other platforms may differ as to requirements here, but in any case the function must be visible to other modules within the same process, but not across processes.

[5] Under Windows, CoGetClassObject, after loading the DLL with CoLoadLibrary, call the Windows API GetProcAddress("DllGetClassObject") to obtain the pointer to the actual function in the DLL.

E_NOINTERFACE	The requested interface was not supported on the class object.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

Note that since DllGetClassObject is passed the CLSID, a single implementation of this function can handle any number of classes. That also means that a single in-process server can implement any number of classes. The implementation of DllGetClassObject only need create the proper class factory for the requested CLSID.

Most implementation of this function for a single class look very much like the implementation of IClassFactory::CreateInstance as illustrated in the code below:

```
HRESULT DllGetClassObject(REFCLSID clsid, REFIID iid, void **ppv) {
    CTextRenderFactory * pCF;
    HRESULT         hr=E_OUTOFMEMORY;

    if (!CLSID_TextRender!=clsid)
        return E_FAIL;
    pCF=new CTextRenderFactory();
    if (NULL==pCF)
        return E_OUTOFMEMORY;

    //This validates the requested interface and calls AddRef
    hr=pCF->QueryInterface(iid, ppv);
    if (FAILED(hr))
        delete pCF;
    else
        ppv=pCF;
    return hr;
}
```

As is conventional with object implementations, including class factories, construction of the object sets the reference count to zero such that the initial QueryInterface creates the first actual reference count. Upon successful return from this function, the class factory will have a reference count of one which must be released by the caller (COM or the client, whoever gets the interface pointer).

The structure of a DLL server with its object and class factory is illustrated in Figure 6-1 below. This figure also illustrates the sequence of calls and events that happen when the client executes the standard object creation sequence of CoGetClassObject and IClassFactory::CreateInstance.

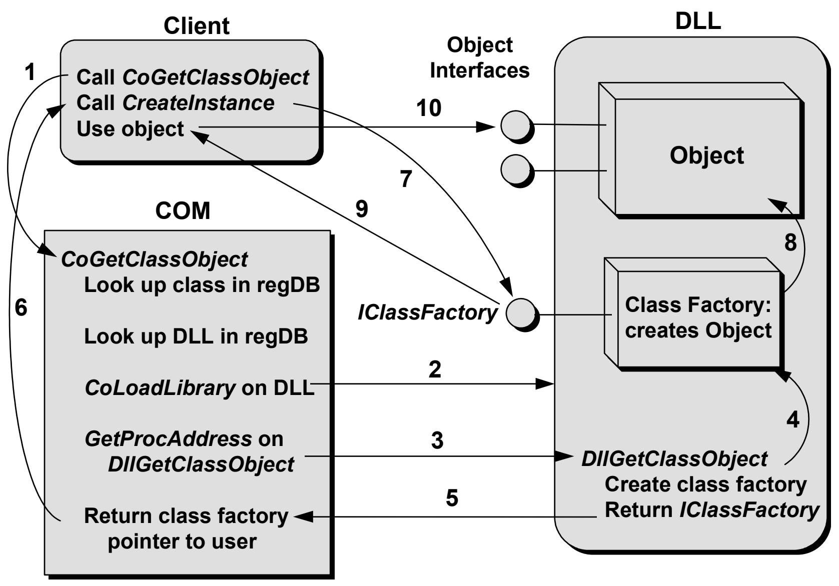

Figure 6-1: Creation sequence of an object from a DLL server. Function calls not in COM are from the Windows API.

1.3.2Exposing the Class Factory from EXE Servers

To expose a class factory from a server application is a different matter than for a DLL server for the reason that the application executes in a different process from the client. Thus, the COM Library cannot just obtain a pointer to an exported function and call that function to retrieve the class factory.

When COM launches an application from within CoGetClassObject it must wait for that application to register a class factory for the desired CLSID through the function CoRegisterClassObject.[6] Once that class factory appears to COM it can return an interface pointer (actually a pointer to the proxy) to the client. CoGetClassObject may time out if the server application takes too long.

The server can differentiate between times it is launched stand-alone and when it is launched from within COM. When COM launches the application it includes a switch "/Embedding"[7] on the server's command line. If the flag is present, the server must register its class factory with CoRegisterClassObject. If the flag is absent, the server may or may not choose to register depending on the object class.

Note that a server application can support any number of object classes by calling CoRegisterClassObject on startup. In fact, a server must register *all* supported class factories because the application is not told which CLSID was requested in the client.

Where CoRegisterClassObject registers a servers factories with COM on startup, the function CoRevokeClassObject unregisters those same factories on application shutdown so they are no longer available, meaning COM must launch the server again for those class factories. Each call to CoRegisterClassObject must be matched with a call to CoRevokeClassObject.

CoRegisterClassObject

HRESULT CoRegisterClassObject(clsid, pUnk, grfContext, grfFlags, pdwRegister)

Registers the specified server class factory identified with *pUnk* with COM in order that it may be connected to by COM Clients. When a server application starts, it creates each class factory it supports and passes them to this function. When a server application exits, it revokes all its registered class objects with CoRevokeClassObject.

[6] This function is called in the COM Library loaded in the server's process space so it actually establishes the remote proxy and stub necessary to perform remote procedure calls.
[7] Case-insensitive. This name originated in OLE 1.0 and has been maintained for such historical reasons and compatibility.

Note that an in-process object could call this function to expose a class factory only when the DLL is already loaded in another process and did not want to expose a class factory until it was loaded for some other reason.

The grfContext flag identifies the execution context of the server and is usually CLSCTX_LOCAL_SERVER. The grfFlags is used to control how connections are made to the class object. Values for this parameter are the following:

```
typedef enum tagREGCLS
    {
    REGCLS_SINGLEUSE = 0,
    REGCLS_MULTIPLEUSE = 1,
    REGCLS_MULTI_SEPARATE = 2
    } REGCLS;
```

Value	Description
REGCLS_SINGLEUSE	Once one client has connected to the class object with CoGetClassObject, then the class object should be removed from public view so that no other clients can similarly connect to it; new clients will use a new instance of the class factory, running a new copy of the server application if necessary. Specifying this flag does not affect the responsibility of the server to call CoRevokeClassObject on shutdown.
REGCLS_MULTIPLEUSE	Many CoGetClassObject calls can connect to the same class factory. When a class factory is registered from a local server (grfContext is CLSCTX_LOCAL_SERVER) and grfFlags includes REGCLS_MULTIPLEUSE, then it is the case that the same class factory will be *automatically also registered* as the in-process server (CLSCTX_IN-PROC_SERVER) for its own process
REGCLS_MULTI_SEPARATE	The same as REGCLS_MULTIPLEUSE, except that registration as a local server does *not* automatically also register as an in-process server in that same process (or any other, for that matter).

Thus, registering as

CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE

is the equivalent to registering as

(CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER), REGCLS_MULTI_SEPARATE

but is different than registering as

CLSCTX_LOCAL_SERVER, REGCLS_MULTI_SEPARATE.

By using REGCLS_MULTI_SEPARATE, an object implementation can cause different class factories to be used according to whether or not it is being created from within the same process as it is implemented.

The following table summarizes the allowable flag combinations and the registrations that are effected by the various combinations:

	REGCLS_-SINGLEUSE	REGCLS_-MULTIPLEUSE	REGCLS_-MULTI_SEPARATE	Other
CLSCTX_IN-PROC_SERVER	*error*	In-Process	In-Process	*error*
CLSCTX_LO-CAL_SERVER	Local	In-Process/Local	**Just Local**	*error*
Both of the above	*error*	In-Process/Local	In-Process/Local	*error*
Other	*error*	*error*	*error*	*error*

The key difference is in the middle columns and the middle rows. In the REGCLS_MULTIPLEUSE column, they are the same (registers multiple use for both InProc and local); in the REGCLS_MULTI_SEPARATE column, the local server case is local only.

The arguments to this function are as follows:

Argument	Type	Description
rclsid	REFCLSID	The CLSID of the class factory being registered.
pUnk	IUnknown *	The class factory whose availability is being published.
grfContext	DWORD	As in CoGetClassObject.
grfFlags	DWORD	REGCLS values that control the use of the class factory.
pdwRegister	DWORD *	A place at which a token is passed back with which this registration can be revoked in CoRevokeClassObject.

Return Value	Meaning
S_OK	Success.
CO_E_OBJISREG	Error. The indicated class is already registered.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

CoRevokeClassObject

HRESULT CoRevokeClassObject(dwRegister)

Informs the COM Library that a class factory previously registered with CoRegisterClassObject is no longer available for use. Server applications call this function on shutdown after having detected the necessary unloading conditions.

- There are no instances of the class in existence, that is, the object count is zero.
- The class factory has a zero number of locks from IClassFactory::LockServer.
- The application servicing the class object is not showing itself to the user (that is, not under user control)

When, subsequently, the reference count on the class object reaches zero, the class object can be destroyed, allowing the application to exit.

Argument	Type	Description
dwRegister	DWORD	A token previously returned from CoRegisterclassObject.

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

The structure of a server application with its object and class factory is illustrated in Figure 6-2. This figure also illustrates the sequence of calls and events that happen when the client executes the standard object creation sequence of CoGetClassObject and IClassFactory::CreateInstance.

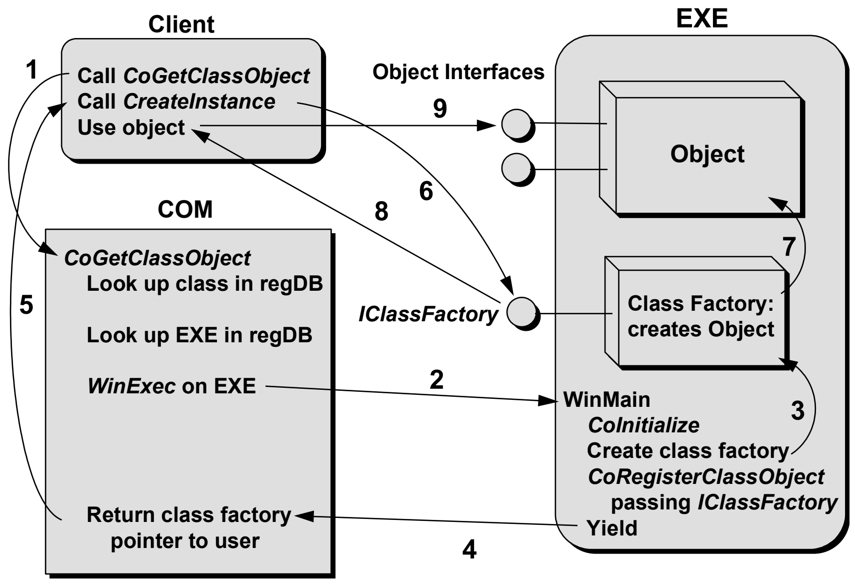

Figure 6-2: Creation sequence of an object from a server application. Function calls not in COM are from the Windows API.

Compare this figure with DLL server Figure 6-1 in the previous section. You'll notice that the structure of the server is generally the same, that is, both have their object and class factory. You'll also notice that the creation sequence from the client's point of view is identical. Again, once the client determines the CLSID of the desired object that client leaves the specifics up to CoGetClassObject. The only differences between the two figures occur inside the COM Library and the specific means of exposing the class factory from the server (along with the unloading mechanism).

Finally, CoRegisterClassObject and CoRevokeClassObject along with when a server calls them demonstrate why a reference count on the class factory is insufficient to keep a server in memory and why IClassFactory::LockServer exists. CoRegisterClassObject must, in order to be implemented properly, hold on to the IUnknown pointer passed to it (that is, the class factory). The reference counting rules state that CoRegisterClassObject must call AddRef on that pointer accordingly. This reference count can only be removed inside CoRevokeClassObject.

However, CoRevokeClassObject is only called on application shutdown and not at any other time. How does the server know when to start its shutdown sequence? Since it has to *be* in the process of shutting down to have the final reference counts on the class factory released through CoRevokeClassObject, it cannot use the reference count to determine when to start the shutdown process in the first place. Therefore there has to be another mechanism through which shutdown is controlled which is IClassFactory::LockServer.

1.4Providing for Server Unloading

When a server has no objects to serve, has no locks, and is not being controlled by an end user (which applies generally to server applications with user interface), then the server has no reason to stay loaded in memory and should provide for unloading itself. This unloading provision differs between server types (DLL and EXE, but no difference for remote servers) as much as class factory registration because whereas a server application can simply terminate itself, an in-process DLL must wait for someone else to explicitly unload it. Therefore the mechanisms for unloading are different and are covered separately in the following sections.

1.4.1Unloading In-Process Servers

As mentioned above, a DLL must wait for someone else to explicitly unload it. The server must, however, have a mechanism through which it indicates whether or not it should be unloaded. That mechanism is a function with the name DllCanUnloadNow that is exported in the same manner as DllGet-ClassObject.

DllCanUloadNow

HRESULT DllCanUnloadNow(void)

DllCanUnloadNow is not provided by COM. Rather, it is a function implemented by and exported from DLLs supporting the Component Object Model. DllCanUnloadNow should be exported from DLLs designed to be dynamically loaded in CoGetClassObject or CoLoadLibrary calls. A DLL is no longer in use when there are no existing instances of classes it manages; at this point, the DLL can be safely freed by calling CoFreeUnusedLibraries. If the DLL loaded by CoGetClassObject fails to export DllCanUnloadNow, the DLL will only be unloaded when CoUninitialize is called to release the COM libraries.

If this function returns S_OK, the duration within which it is in fact safe to unload the DLL depends on whether the DLL is single or multi-thread aware. For single thread DLLs, it is safe to unload the DLL up until such time as the thread on which DllCanUnloadNow was invoked causes it to be otherwise (objects created, for example).

Return Value	Meaning
S_OK	The DLL may be unloaded now.
S_FALSE	The DLL should not be unloaded at the present time.

1.4.2 Unloading EXE Servers

A server application is responsible for unloading itself, simply by terminating and exiting its main entry function[8], when the shutdown conditions are met, including whether or not the user has control. In the ongoing example of this chapter, this would involve detecting the proper shutdown conditions whenever an object is destroyed (in the suggested ObjectDestroyed function) or whenever the last lock is removed (in IClassFactory::LockServer).

```
//User control flag
BOOL        g_fUser=FALSE;

void ObjectDestroyed(void) {
      g_cObj--;
      if (0L==g_cObj && 0L==g_cLock && !g_fUser)
            //Begin shutdown
      return;
      }

HRESULT CTextRenderFactory::LockServer(BOOL fLock) {
      if (fLock)
            g_cLock++;              // for single threaded app only, of course
      else {
            g_cLock--;
            if (0L==g_cObj && 0L==g_cLock && !g_fUser)
                  //Begin shutdown
            }
      return NOERROR;
      }
```

If desired, you can of course centralize the shutdown conditions by artificially incrementing the object count in IClassFactory::LockServer and directly calling ObjectDestroyed. That way you do not need redundant code in both functions.

During shutdown, the server is responsible for calling CoRevokeClassObject on all previously registered class factories and for calling CoUninitialize like any COM application.

A server application only needs a "user-control" flag if it becomes visible in some way and also allows the user to perform some action which would necessitate the application stays running regardless of any other conditions. For example, the server might be running to service an object for a client and the user opens another file in that same application. Since the user is the only agent who can close the file, the

[8] Under Microsoft Windows, the application usually starts shutdown by posting a WM_CLOSE message to its main window, simulating what happens when a user closes an application. This eventually causes the application to exit the WinMain function.

user control flag is set to TRUE meaning that the user must explicitly close the application: no automatic shutdown is possible.

If a server is visible and under user control, there is the possibility that clients have connections to objects within that server when the user explicitly closes the application. In that situation the server can take one of two actions:

5. Simply hide the application and reset the user control flag to FALSE such that the server will automatically shut down when all objects and locks are released.
6. Terminate the application but call CoDisconnectObject for each object in service to forcibly disconnect all clients.

The second option, though more brutal, is necessary in some situations. The CoDisconnectObject function exists to insure that all external reference counts to the server's objects are released such that the server can release its own references and destroy all objects.

CoDisconnectObject

HRESULT CoDisconnectObject(pUnk, dwReserved)

This function serves any extant remote connections that are being maintained on behalf of all the interface pointers on this object. This is a very rude and privileged operation which should generally only be invoked by the process in which the object actually is managed by the object implementation itself.

The primary purpose of this operation is to give an application process certain and definite control over connections to other processes that may have been made from objects managed by the process. If the application process wishes to exit, then we do not want it to be the case that the extant reference counts from clients of the application's objects in fact keeps the process alive. The process can call this function for each of the objects that it manages without waiting for any confirmation from clients. Having thus released resources maintained by the remoting connections, the application process can exit safely and cleanly. In effect, CoDisconnectObject causes a controlled crash of the remoting connections to the object.

Argument	Type	Description
pUnk	IUnknown *	The object that we wish to disconnect. May be any interface on the object which is polymorphic with IUnknown, not necessarily the exact interface returned by QueryInterface(IID_IUnknown...).
dwReserved	DWORD	Reserved for future use; must be zero.

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

1.5Object Handlers

As mentioned earlier this specification, object handlers from one perspective are special cases of in-process servers that talk to their local or remote servers as well as a client. From a second perspective, an object handler is really just a fancy proxy for a local or remote server that does a little more than just forward calls through RPC. The latter view is more precise architecturally: a "handler" is simply the piece of code that runs in the client's space on behalf of a remote object; it can be used synonymously with the term "proxy object." The handler may be a trivial one, one that simply forwards all of its calls on to the remote object, or it may implement some amount of non-trivial client side processing. (In practice, the term "proxy object" is most often reserved for use with trivial handlers, leaving "handler" for the more general situation.)

The structure of an object handler is exactly the same as a full-in process server: an object handler implements an object, a class factory, and the two functions DllGetClassObject and DllCanUnloadNow exactly as described above.

The key difference between handlers and full DLL servers (and simple proxy objects, for that matter) is the extent to which they implement their respective objects. Whereas the full DLL server implements the complete object (using other objects internally, if desired), the handler only implements a partial object depending on a local or remote server to complete the implementation. Again, the reasons for this is that sometimes a certain interface can only be useful when implemented on an in-process object, such as when member functions of that interface contain parameters that cannot be shared between processes. Thus the object in the handler would implement the restricted in-process interface but leave all others for implementation in the local or remote server.

1.6Object Reusability

With object-oriented programming it is often true that there already exists some object that implements some of what you want to implement, and instead of rewriting all that code yourself you would like to reuse that other object for your own implementation. Hence we have the desire for object reusability and a number means to achieve it such as implementation inheritance, which is exploited in C++ and other languages. However, as discussed in the "Object Reusability" section of Chapter 2, implementation inheritance has some significant drawbacks and problems that do not make it a good object reusability mechanism for a system object model.

For that reason COM supports two notions of object reuse, containment and aggregation, that were also described in Chapter 2. In that chapter we saw that containment, the most common and simplest for of object reuse, is where the "outer object" simply uses other "inner objects" for their services. The outer object is nothing more than a client of the inner objects. We also saw in Chapter 2 the notion of aggregation, where the outer object exposes interfaces from inner objects as if the outer object implemented those interfaces itself. We brought up the catch that there has to be some mechanism through which the IUnknown behavior of inner object interfaces exposed in this manner is appropriate to the outer object. We are now in a position to see exactly how the solution manifests itself.

The following sections treat Containment and Aggregation in more detail using the TextRender object as an example. To refresh our memory of this object's purpose, the following list reiterates the specific features of the TextRender object that implements the IPersistFile and IDataObject interfaces:

- Read text from a file through IPersistFile::Load
- Write text to a file through IPersistFile::Save
- Accept a memory copy of the text through IDataObject::SetData
- Render a memory copy of the text through IDataObject::GetData
- Render metafile and bitmap images of the text also through IDataObject::GetData

1.6.1Reusability Through Containment

Let's say that when we decide to implement the TextRender object we find that another object exists with CLSID_TextImage that is capable of accepting text through IDataObject::SetData but can do nothing more than render a metafile or bitmap for that text through IDataObject::GetData. This "TextImage" object cannot render memory copies of the text and has no concept of reading or writing text to a file. But it does such a good job implementing the graphical rendering that we wish to use it to help implement our TextRender object.

In this case the TextRender object, when asked for a metafile or bitmap of its current text in IDataObject::GetData, would delegate the rendering to the TextImage object. TextRender would first call TextImage's IDataObject::SetData to give it the most recent text (if it has changed since the last call) and then call TextImage's IDataObject::GetData asking for the metafile or bitmap format. This delegation is illustrated in Figure 6-3.

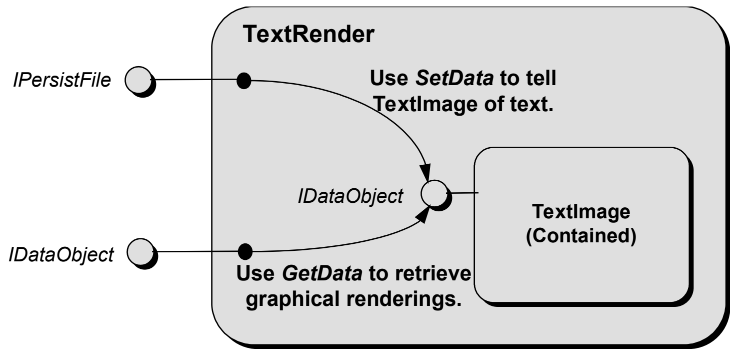
TextRender
IPersistFile
Use *SetData* to tell TextImage of text.
IDataObject
TextImage (Contained)
IDataObject
Use *GetData* to retrieve graphical renderings.

Figure 6-3: An outer object that uses inner objects through containment is a client of the inner objects.

To create this configuration, the TextRender object would, during its own creation, instantiate the TextImage object with the following code, storing the TextImage's IDataObject pointer in a TextImage field m_pIDataObjImage:

```
//TextRender initialization
HRESULT    hr;
hr=CoCreateInstance(CLSID_TextImage, CLSCTX_SERVER, NULL, IID_IDataObject, (void *)&m_pIDataObjImage);
if (FAILED(hr))
    //TextImage not available, either fail or disable graphic rendering
//Success:  can now make use of TextImage object.
```

This code is included here to show the NULL parameter in the middle of the call to CoCreateInstance. This is the "outer unknown" and is only applicable to aggregation. Containment does not make use of the outer unknown concept and so this parameter should always be NULL.

Now that the TextRender object has TextImage's IDataObject it can delegate functionality to TextImage as needed. The following pseudo-code illustrates how TextRender's IDataObject::GetData function might be implemented:

```
HRESULT CTextRender::GetData(FORMATETC *pFE, STGMEDIUM *pSTM)
    {
    switch ([format in FORMATETC])
        {
        case <text>:
            //Make copy of text and return
        case <metafile>:
        case <bitmap>:
            //Insure TextImage has current text
            m_pIDataObjImage->SetData(<copy of our current text>);
            return m_pIDataObjImage->GetData(pFE, pSTM);
        }
    return <error>;
    }
```

Note that if the TextImage object was modified at some later date to implement additional interfaces (such as IPersistFile) or was updated to also support rendering copies of text in memory just like TextRender, **the code above would still function perfectly.** This is the *key* power of COM's reusability mechanisms over traditional language-style implementation inheritance: the reused object can freely revise itself so long as it continues to provide the exact behavior it has provided in the past. Since the TextRender object never bothers to query for any other interface on TextImage, and because it never call's TextImage's GetData for any format other than metafile or bitmap, TextImage can implement any number of new interfaces and support any number of new formats in GetData. All TextImage has to insure is that the behavior of SetData for text and the behavior of GetData for metafiles and bitmaps remains the same.

Of course, this is just a simple example of containment. Real components will generally be much more complex and will generally make use of many inner objects and many more interfaces in this manner. But again, since the outer object only depends on the *behavior* of the inner object and does not care how it goes about performing its operations, the inner object can be modified without requiring any recompilation or any other changes to the outer object. That is reusability at its finest.

1.6.2Reusability Through Aggregation

Let's now say that we are planning to revise our TextRender object at a later time than out initial containment implementation in the previous section. At that time we find that the implementor of the TextImage object at the time the implementor of the TextRender object sat down to work (or perhaps is making a revision of his object) that the vendor of the TextImage object has improved TextImage such that it implements everything that TextRender would like to do through its IDataObject interface. That is, TextImage still accepts text through SetData but has recently added the ability to make copies of its text and provide those copies through GetData in addition to metafiles and bitmaps.

In this case, the implementor of TextRender now sees that TextImage's implementation of IDataObject is exactly the implementation that TextRender requires. What we, as the implementors of TextRender, would like to do now is simply expose TextImage's IDataObject as our own as shown in Figure 6-4.

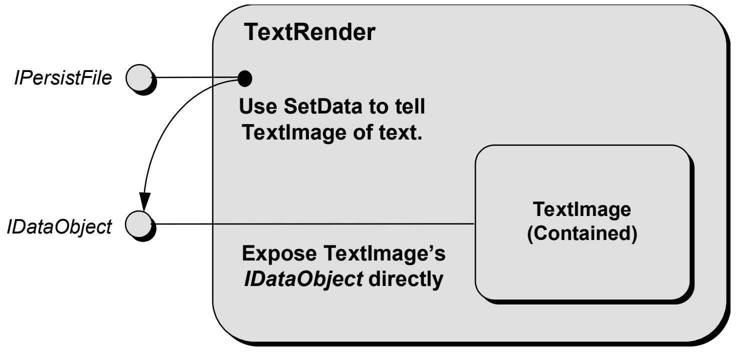
TextRender
IPersistFile
Use SetData to tell TextImage of text.
IDataObject
TextImage (Contained)
Expose TextImage's *IDataObject* directly

Figure 6-4: When an inner object does a complete job implementing an interface, outer objects may want to expose the interface directly.

The only catch is that we must implement the proper behavior of the IUnknown members in the inner object's (TextImage) IDataObject interface: AddRef and Release have to affect the reference count on the outer object (TextRender) and not the reference count of the inner object. Furthermore, QueryInterface has to be able to return the TextRender object's IPersistFile interface. The solution is to inform the inner object that it is being used in an aggregation such that when it sees IUnknown calls to its interfaces it can delegate those calls to the outer object.

One other catch remains: the outer object must have a means to control the lifetime of the inner object through AddRef and Release as well as have a means to query for the interfaces that only exist on the inner object. For that reason, the inner object *must implement an isolated version of IUnknown that controls the inner object exclusively and never delegates to the outer object.*[9] This requires that the inner object separates the IUnknown members of its functional interfaces from an implementation of IUnknown that strictly controls the inner object itself. In other words, the inner object, to support aggregation, must implement two sets of IUnknown functions: delegating and non-delegating.

This, then, is the mechanism for making aggregation work:

1. When creating the inner object, the outer object must pass its own IUnknown to the inner object through the pUnkOuter parameter of IClassFactory::CreateInstance. pUnkOuter in this case is called the "controlling unknown."
2. The inner object must check pUnkOuter in its implementation of CreateInstance. If this parameter is non-NULL, then the inner object knows it is being created as part of an aggregate. If the inner object does not support aggregation, then it must fail with CLASS_E_NOAGGREGATION. If aggregation is supported, the inner object saves pUnkOuter for later use, but does not call AddRef on it. The reason is that the inner object's lifetime is entirely contained within the outer object's lifetime, so there is no need for the call and to do so would create a circular reference.

[9] An interface with such an IUnknown is sometimes called an "inner" interface on the aggregated object. There may, in general, be several inner interfaces on an object. IRpcProxyBuffer, for example, is one. This is a property of the interface itself, not the implementation.

3. If the inner object detects a non-NULL pUnkOuter in CreateInstance, and the call requests the interface IUnknown itself (as is almost always the case), the inner object must be sure to return its non-delegating IUnknown.
4. If the inner object itself aggregates other objects (which is unknown to the outer object) it must pass the same pUnkOuter pointer it receives down to the next inner object.
5. When the outer object is queried for an interface it exposes from the inner object, the outer object calls QueryInterface in the non-delegating IUnknown to obtain the pointer to return to the client.
6. The inner object must delegate to the controlling unknown, that is, pUnkOuter, all IUnknown calls occurring in any interface it implements other than the non-delegating IUnknown.

Through these steps, the inner object is made aware of the outer object, obtains an IUnknown to which it can delegate calls to insure proper behavior of reference counting and QueryInterface, and provides a way for the outer object to control the inner object's lifetime separately. The mechanism is illustrated in Figure 6-5.

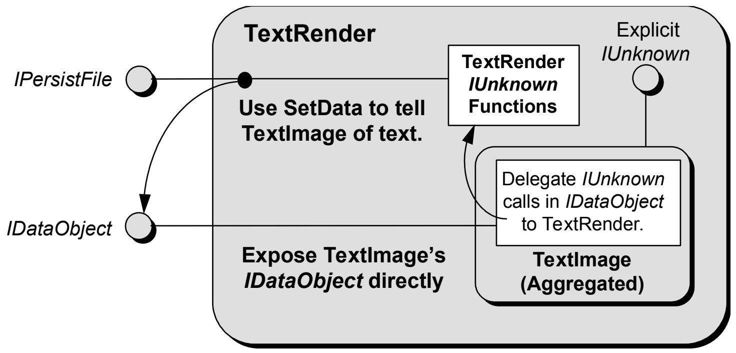
TextRender
Explicit *IUnknown*
IPersistFile
TextRender *IUnknown* Functions
Use SetData to tell TextImage of text.
Delegate *IUnknown* calls in *IDataObject* to TextRender.
IDataObject
Expose TextImage's *IDataObject* directly
TextImage (Aggregated)

Figure 6-5: Aggregation requires an explicit implementation of *IUnknown* on the inner object and delegation of *IUnknown* function of any other interface to the outer object's *IUnknown* functions.

Now let's look at how this mechanism manifests in code. First off, the TextRender object no longer needs it's own IDataObject implementation and can thus remove it from it's class, but will need to add a member m_pUnkImage to maintain the TextImage's non-delegating *IUnknown*:

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

        IUnknown *     m_pUnkImage;         //TextImage IUnknown
        //Other internal member functions here

    public:
        [Constructor, Destructor]

        //Outer object IUnknown
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
        ULONG Release(void);

        //IPersistFile Member overrides
        ...
    };
```

In the previous section we saw how the TextRender object would create a TextImage object for containment using CoCreateInstance with the pUnkOuter parameter set to NULL. In aggregation, this parameter will be TextRender's own IUnknown (obtained using a typecast). Furthermore, TextRender must request IUnknown initially from TextImage (storing the pointer in m_pUnkImage*)*:

```
//TextRender initialization
HRESULT    hr;
hr=CoCreateInstance(CLSID_TextImage, CLSCTX_ SERVER, (IUnknown *)this, IID_IUnknown, (void *)&m_pUnkImage);
```

```
if (FAILED(hr))
        //TextImage not available, either fail or disable graphic rendering
//Success:  can now make use of TextImage object.
```

Now, since TextRender does not have it's own IDataObject any longer, its implementation of QueryInterface will use m_pUnkImage to obtain interface 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)
                return m_pUnkImage->QueryInterface(iid, ppv);

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

        //Any call to anyone's AddRef is our own.
        AddRef();
        return NOERROR;
        }
```

Note that delegating QueryInterface to the inner object is done only for those interfaces that the outer object knows it wants to expose. The outer object should not delegate the query as a default case, for such blind forwarding without an understanding of the semantic being forwarded will almost assuredly break the outer object should the inner one be revised with new functionality.

Caching interfaces on the inner object

In order to avoid reference counting cycles, special action is needed if the outer object wishes to cache pointers to the inner object's interfaces.

Specifically, if the outer object wishes to cache a to an inner object's interface, once it has obtained the interface from the inner object, the outer object should invoke Release on the punkOuter that was given to the inner object at its instantiation time.

```
// Obtaining inner object interface pointer
pUnkInner->QueryInterface(IID_IFoo, &pIFoo);
pUnkOuter->Release();

// Releasing inner object interface pointer
pUnkOuter->AddRef();
pIFoo->Release();
```

It is suggested that to allow inner objects to do better resource management that controlling objects delay the acquisition of cached pointers and release them when there is no possible use for them.

Efficiency at any Depth of Aggregation

Aggregation has one interesting aspect when aggregates are used on more than one level of an object implementation. Imagine that the TextImage object in the previous example is itself an aggregate object that uses other inner objects. In such a case TextImage will be passing some controlling unknown to those other inner objects. If TextImage is not being aggregated by anyone else, then the controlling unknown is its own; otherwise it passes the pUnkOuter from IClassFactory::CreateInstance on down the line, and any other inner objects that are aggregates themselves do the same.

The net result is that any object in an aggregation, no matter how deeply it is buried in the overall structure, will almost always delegate directly to the controlling unknown if it's interface is exposed from that final outer object. Therefore performance and efficiency of multiple levels of aggregation is not an issue. At worst each delegation is a single extra function call.

1.7Emulating Other Servers

The final topic related to COM Servers for this chapter is what is known as emulation: the ability for one server associated with one CLSID to emulate a server of another CLSID. A server that can emulate another is responsible for providing compatible behavior for a different class through a different implementation. This forms the basis for allowing end-users the choice in which servers are used for which objects, as long as the behavior is compatible between those servers.

As far as COM is concerned, it only has to provide some way for a server to indicate that it wishes to emulate some CLSID. To that end, the COM Library supplies the function CoTreatAsClass to establish an emulation that remains in effect (persistently) until canceled or changed. In addition it supplies CoGetTreatAsClass to allow a caller to determine if a given CLSID is marked for emulation.

CoTreatAsClass

HRESULT CoTreatAsClass(clsidOld, clsidNew)

Establish or cancel an emulation relationship between two classes. When clsidNew is emulating clsidOld, calls to CoGetClassObject with clsidOld will transparently use clsidNew. Thus, for example, creating an object of clsidOld will in fact launch the server for clsidNew and have it create the object instead.

This function does no validation on whether an appropriate registration entries exist for clsidNew.

An emulation is canceled by calling this function with clsidOld equal to the original class and clsidNew set to CLSID_NULL.

Argument	Type	Description
clsidOld	REFCID	The class to be emulated.
clsidNew	REFCID	The class which should emulate clsidOld. This replaces any existing emulation for clsidOld. May be CLSID_NULL, in which case any existing emulation for clsidOld is removed.

Return Value	Meaning
S_OK	Success.
CO_E_CLASSNOTREG	*to be described.*
CO_E_READREGDB	*to be described.*
CO_E_WRITEREGDB	*to be described.*
E_UNEXPECTED	An unspecified error occurred.

CoGetTreatAsClass

HRESULT CoGetTreatAsClass(clsidOld, pclsidNew)

Return the existing emulation information for a given class. If no emulation entry exists for clsidOld then clsidOld is returned in pclsidNew.

Argument	Type	Description
clsidOld	REFCID	The class for which the emulation information is to be retrieved.
pclsidNew	CLSID *	The place at which to return the class, if any, which emulates clsidOld. clsidOld is returned if there is no such class. pclsidNew may not be NULL.

Return Value	Meaning
S_OK	Success. A new, (possibly) different CLSID is returned through *pclisdNew.
S_FALSE	Success. The class is emulating itself.
CO_E_READREGDB	.
E_UNEXPECTED	An unspecified error occurred.

How the COM Library implements these functions depends upon the structure of the system registry. For example, under Microsoft Windows, COM uses an additional subkey under an object's CLSID key in the form of:

```
    TreatAs = {<new CLSID>}
```

When the Windows' COM implementation of CoGetClassObject attempts to locate a server for a CLSID, it will always call CoGetTreatAsClass to retrieve the actual CLSID to use. Since CoGetTreatAsClass will return the same CLSID as passed in if no emulation exists, COM doesn’t have to do any special case checks for emulation.

*This page intentionally left blank.**This page intentionally left blank.*