Introduction to COM Part II - Behind the Scenes of a COM Server

By Michael Dunn

Posted: **11 Jan 2001**
Updated: **3 Apr 2001**
Views: **411,168**
Bookmarked: **299 times**

A tutorial for programmers new to COM that explains the internals of COM servers, and how to write your own interfaces in C++

179 votes for this article.
Popularity: 10.90 Rating: **4.84** out of 5

Download source files - 25 Kb
Download demo project - 37 Kb

Purpose of this Article

As with my first Introduction to COM article, I have written this tutorial for programmers who are just starting out in COM and need some help in understanding the basics. This article covers COM from the server side of things, explaining the steps required to write your own COM interfaces and COM servers, as well as detailing what exactly happens in a COM server when the COM library calls into it.

Introduction

If you've read my first Intro to COM article, you should be well-versed in what's involved in using COM as a client. Now it's time to approach COM from the other side - the COM server. I'll cover how to write a COM server from scratch in plain C++, with no class libraries involved. While this isn't necessarily the approach usually taken nowadays, seeing all the code that goes into making a COM server - with nothing hidden away in a pre-built library - is really the best way to fully understand everything that happens in the server.

This article assumes you are proficient in C++ and understand the concepts and terminology covered in the first Intro to COM article. The sections in the article are:

Quick Tour of a COM Server - Describes the basic requirements of a COM server.

Server Lifetime Management - Describes how a COM server controls how long it remains loaded.

Implementing Interfaces, Starting With IUnknown - Shows how to write an implementation of an interface in a C++ class, and describes the purpose of the `IUnknown` methods.

Inside CoCreateInstance() - An overview of what happens when you call `CoCreateInstance()`.

COM Server Registration - Describes the registry entries needed to properly register a COM server.

Creating COM Objects - The Class Factory - Describes the process of creating COM objects for your client program to use.

A Sample Custom Interface - Some sample code that illustrates the concepts from the previous sections.

A Client to Use Our Server - Demonstrates a simple client app we can use to test our server.

Other Details - Notes on the source code and debugging.

Quick Tour of a COM Server

In this article, we'll be looking at the simplest type of COM server, an *in-process server*. "In-process" means that the server is loaded into the process space of the client program. In-process (or "in-proc") servers are always DLLs, and must be on the same computer as the client program.

An in-proc server must meet two criteria before it can be used by the COM library:

1. It must be registered properly under the HKEY_CLASSES_ROOT\CLSID key.
2. It must export a function called DllGetClassObject().

This is the bare minimum you need to do to get an in-proc server working. A key with the server's GUID as its name must be created under the HKEY_CLASSES_ROOT\CLSID key, and that key must contain a couple of values listing the server's location and its threading model. The DllGetClassObject() function is called by the COM library as part of the work done by the CoCreateInstance() API.

There are three other functions that are usually exported as well:

- DllCanUnloadNow(): Called by the COM library to see if the server may be unloaded from memory.
- DllRegisterServer(): Called by an installation utility like RegSvr32 to let the server register itself.
- DllUnregisterServer(): Called by an uninstallation utility to remove the registry entries created by DllRegisterServer().

Of course, it's not enough to just export the right functions - they have to conform to the COM spec so that the COM library and the client program can use the server.

Server Lifetime Management

One unusual aspect of DLL servers is that they control how long they stay loaded. "Normal" DLLs are passive and are loaded/unloaded at the whim of the application using them. Technically, DLL servers are passive as well, since they are DLLs after all, but the COM library provides a mechanism that allows a server to instruct COM to unload it. This is done through the exported function DllCanUnloadNow(). The prototype for this function is:

```
HRESULT DllCanUnloadNow();
```

When the client app calls the COM API CoFreeUnusedLibraries(), usually during its idle processing, the COM library goes through all of the DLL servers that the app has loaded and queries each one by calling its DllCanUnloadNow() function. If a server needs to remain loaded, it returns S_FALSE. On the other hand, if a server determines that it no longer needs to be in memory, it can return S_OK to have COM unload it.

The way a server tells if it can be unloaded is a simple reference count. An implementation of DllCanUnloadNow() might look like this:

```
extern UINT g_uDllRefCount;  // server's reference count

HRESULT DllCanUnloadNow()
{
    return (g_uDllRefCount > 0) ? S_FALSE : S_OK;
}
```

I will cover how the reference count is maintained in the next section, once we get to some sample code.

Implementing Interfaces, Starting With IUnknown

Recall that every interface derives from IUnknown. This is because IUnknown covers two basic features of COM objects - reference counting and interface querying. When you write a coclass, you also write an implementation of IUnknown that meets your needs. Let's take as an example a coclass that just implements IUnknown -- the simplest possible coclass you could write. We will implement IUnknown in a C++ class called CUnknownImpl. The class declaration looks like this:

```
class CUnknownImpl : public IUnknown
{
public:
    // Construction and destruction
    CUnknownImpl();
    virtual ~CUnknownImpl();

    // IUnknown methods
    ULONG AddRef();
    ULONG Release)();
    HRESULT QueryInterface( REFIID riid, void** ppv );
```

```
protected:
    UINT m_uRefCount;  // object's reference count
};
```

The constructor and destructor

The constructor and destructor manage the server's reference count:

```
CUnknownImpl::CUnknownImpl()
{
    m_uRefCount = 0;
    g_uDllRefCount++;
}

CUnknownImpl::~CUnknownImpl()
{
    g_uDllRefCount--;
}
```

The constructor is called when a new COM object is created, so it increments the server's reference count to keep the server in memory. It also initializes the *object's* reference count to zero. When the COM object is destroyed, it decrements the server's reference count.

AddRef() and Release()

These two methods control the lifetime of the COM object. `AddRef()` is simple:

```
ULONG CUnknownImpl::AddRef()
{
    return ++m_uRefCount;
}
```

`AddRef()` simply increments the object's reference count, and returns the updated count.

`Release()` is a bit less trivial:

```
ULONG CUnknownImpl::Release()
{
ULONG uRet = --m_uRefCount;

    if ( 0 == m_uRefCount )  // releasing last reference?
        delete this;

    return uRet;
}
```

In addition to decrementing the object's reference count, `Release()` destroys the object if it has no more outstanding references. `Release()` also returns the updated reference count. Notice that this implementation of `Release()` assumes that the COM object was created on the heap. If you create an object on the stack or at global scope, things will go awry when the object tries to delete itself.

Now it should be clear why it's important to call `AddRef()` and `Release()` properly in your client apps! If you don't call them correctly, the COM objects you're using may be destroyed too soon, or not at all. And if COM objects get destroyed too soon, that can result in an entire COM server being yanked out of memory, causing your app to crash the next time it tries to access code that was in that server.

If you've done any multithreaded programming, you might be wondering about the thread-safety of using `++` and `--` instead of `InterlockedIncrement()` and `InterlockedDecrement()`. `++` and `--` are perfectly safe to use in single-threaded servers, because even if the client app is multi-threaded and makes method calls from different threads, the COM library serializes method calls into our server. That means that once one method call begins, all other threads attempting to call methods will block until the first method returns. The COM library itself ensures that our server will never be entered by more than one thread at a time.

QueryInterface()

QueryInterface(), or QI() for short, is used by clients to request different interfaces from one COM object. Since our sample coclass only implements one interface, our QI() will be easy. QI() takes two parameters: the IID of the interface being requested, and a pointer-sized buffer where QI() stores the interface pointer if the query is successful.

```
HRESULT CUnknownImpl::QueryInterface ( REFIID riid, void** ppv )
{
HRESULT hrRet = S_OK;

    // Standard QI() initialization - set *ppv to NULL.
    *ppv = NULL;

    // If the client is requesting an interface we support, set *ppv.
    if ( IsEqualIID ( riid, IID_IUnknown ))
        {
        *ppv = (IUnknown*) this;
        }
    else
        {
        // We don't support the interface the client is asking for.
        hrRet = E_NOINTERFACE;
        }

    // If we're returning an interface pointer, AddRef() it.
    if ( S_OK == hrRet )
        {
        ((IUnknown*) *ppv)->AddRef();
        }

    return hrRet;
}
```

There are three different things done in QI():

1. Initialize the passed-in pointer to NULL. [*ppv = NULL;]
2. Test riid to see if our coclass implements the interface the client is asking for. [if (IsEqualIID (riid, IID_IUnknown))]
3. If we do implement the requested interface, increment the COM object's reference count. [((IUnknown*) *ppv)->AddRef();]

Note that the AddRef() is critical. This line:

```
    *ppv = (IUnknown*) this;
```

creates a new reference to the COM object, so we must call AddRef() to tell the object that this new reference exists. The cast to IUnknown* in the AddRef() call may look odd, but in a non-trivial coclass' QI(), *ppv may be something other than an IUnknown*, so it's a good idea to get in the habit of using that cast.

Now that we've covered some internal details of DLL servers, let's step back and see how our server is used when a client calls CoCreateInstance().

Inside CoCreateInstance()

Back in the first Intro to COM article, we saw the CoCreateInstance() API, which creates a COM object when a client requests one. From the client's perspective, it's a black box. Just call CoCreateInstance() with the right parameters and *BAM!* you get a COM object back. Of course, there's no black magic involved; a well-defined process happens in which the COM server gets loaded, creates the requested COM object, and returns the requested interface.

Here's a quick overview of the process. There are a few unfamiliar terms here, but don't worry; I'll cover everything in the following sections.

1. The client program calls CoCreateInstance(), passing the CLSID of the coclass and the IID of the interface it wants.

2. The COM library looks up the server's CLSID under `HKEY_CLASSES_ROOT\CLSID`. This key holds the server's registration information.
3. The COM library reads the full path of the server DLL and loads the DLL into the client's process space.
4. The COM library calls the `DllGetClassObject()` function in the server to request the class factory for the requested coclass.
5. The server creates a class factory and returns it from `DllGetClassObject()`.
6. The COM library calls the `CreateInstance()` method in the class factory to create the COM object that the client program requested.
7. `CoCreateInstance()` returns an interface pointer back to the client program.

COM Server Registration

For anything else to work, a COM server must be properly registered in the Windows registry. If you look at the `HKEY_CLASSES_ROOT\CLSID` key, you'll see a *ton* of subkeys. `HKCR\CLSID` holds a list of every COM server available on the computer. When a COM server is registered (usually via `DllRegisterServer()`), it creates a key under the `CLSID` key whose name is the server's GUID in standard registry format. An example of registry format is:

```
{067DF822-EAB6-11cf-B56E-00A0244D5087}
```

The braces and hyphens are required, and letters can be either upper- or lower-case.

The default value of this key is a human-readable name for the coclass, which should be suitable for display in a UI by tools like the OLE/COM Object Viewer that ships with VC.

More information can be stored in subkeys under the GUID key. Which subkeys you need to create depends greatly on what type of COM server you have, and how it can be used. For the purposes of our simple in-proc server, we only need one subkey: `InProcServer32`.

The `InProcServer32` key contains two strings: the default value, which is the full path to the server DLL; and a `ThreadingModel` value that holds (what else?) threading model. Threading models are beyond the scope of this article, but suffice it to say that for single-threaded servers, the model to use is `Apartment`.

Creating COM Objects - The Class Factory

Back when we were looking at the client side of COM, I talked about how COM has its own language-independent procedures for creating and destroying COM objects. The client calls `CoCreateInstance()` to create a new COM object. Now, we'll see how it works on the server side.

Every time you implement a coclass, you also write a companion coclass which is responsible for creating instances of the first coclass. This companion is called the *class factory* for the coclass and its sole purpose is to create COM objects. The reason for having a class factory is language-independence. COM itself doesn't create COM objects, because that wouldn't be language- and implementation-independent.

When a client wants to create a COM object, the COM library requests the class factory from the COM server. The class factory then creates the COM object which gets returned to the client. The mechanism for this communication is the exported function `DllGetClassObject()`.

> A quick sidebar is in order here. The terms "class factory" and "class object" actually refer to the same thing. However, neither term accurately describes the purpose of the class factory, since the factory creates COM *objects*, not COM classes. It may help you to mentally replace "class factory" with "object factory." (In fact, MFC did this for real - its class factory implementation is called `COleObjectFactory`.) However, the official term *is* "class factory," so that's what I'll use in this article.

When the COM library calls `DllGetClassObject()`, it passes the CLSID that the client is requesting. The server is responsible for creating the class factory for the requested CLSID and returning it. A class factory is itself a coclass, and implements the `IClassFactory` interface. If `DllGetClassObject()` succeeds, it returns an `IClassFactory` pointer to the COM library, which then uses `IClassFactory` methods to create an instance of the COM object the client requested.

The `IClassFactory` interface looks like this:

```
struct IClassFactory : public IUnknown
{
    HRESULT CreateInstance( IUnknown* pUnkOuter, REFIID riid,
                            void** ppvObject );
    HRESULT LockServer( BOOL fLock );
};
```

`CreateInstance()` is the method that creates new COM objects. `LockServer()` lets the COM library increment or decrement the server's reference count when necessary.

A Sample Custom Interface

For an example of class factories at work, let's start taking a look at the article's sample project. It's a DLL server that implements an interface `ISimpleMsgBox` in a coclass called `CSimpleMsgBoxImpl`.

The interface definition

Our new interface is called `ISimpleMsgBox`. As with all interfaces, it must derive from `IUnknown`. There's just one method, `DoSimpleMsgBox()`. Note that it returns the standard type `HRESULT`. All methods you write should have `HRESULT` as the return type, and any other data you need to return to the caller should be done through pointer parameters.

```
struct ISimpleMsgBox : public IUnknown
{
    // IUnknown methods
    ULONG AddRef();
    ULONG Release();
    HRESULT QueryInterface( REFIID riid, void** ppv );

    // ISimpleMsgBox methods
    HRESULT DoSimpleMsgBox( HWND hwndParent, BSTR bsMessageText );
};

struct __declspec(uuid("{7D51904D-1645-4a8c-BDE0-0F4A44FC38C4}"))
                  ISimpleMsgBox;
```

(The `__declspec` line assigns a GUID to the `ISimpleMsgBox` symbol, and that GUID can later be retrieved with the `__uuidof` operator. Both `__declspec` and `__uuidof` are Microsoft C++ extensions.)

The second parameter of `DoSimpleMsgBox()` is of type `BSTR`. `BSTR` stands for "binary string" - COM's representation of a fixed-length sequence of bytes. `BSTR`s are used mainly by scripting clients like Visual Basic and the Windows Scripting Host.

This interface is then implemented by a C++ class called `CSimpleMsgBoxImpl`. Its definition is:

```
class CSimpleMsgBoxImpl : public ISimpleMsgBox
{
public:
        CSimpleMsgBoxImpl();
        virtual ~CSimpleMsgBoxImpl();

    // IUnknown methods
    ULONG AddRef();
    ULONG Release();
    HRESULT QueryInterface( REFIID riid, void** ppv );

    // ISimpleMsgBox methods
    HRESULT DoSimpleMsgBox( HWND hwndParent, BSTR bsMessageText );

protected:
    ULONG m_uRefCount;
};

class  __declspec(uuid("{7D51904E-1645-4a8c-BDE0-0F4A44FC38C4}"))
                  CSimpleMsgBoxImpl;
```

When a client wants to create a `SimpleMsgBox` COM object, it would use code like this:

```
ISimpleMsgBox* pIMsgBox;
HRESULT hr;

hr = CoCreateInstance( __uuidof(CSimpleMsgBoxImpl), // CLSID of the coclass
                       NULL,                          // no aggregation
                       CLSCTX_INPROC_SERVER,          // the server is in-proc
                       __uuidof(ISimpleMsgBox),       // IID of the interface
                                                      // we want
                       (void**) &pIMsgBox );          // address of our
                                                      // interface pointer
```

The class factory

Our class factory implementation

Our SimpleMsgBox class factory is implemented in a C++ class called, imaginatively enough, CSimpleMsgBoxClassFactory:

```
class CSimpleMsgBoxClassFactory : public IClassFactory
{
public:
    CSimpleMsgBoxClassFactory();
    virtual ~CSimpleMsgBoxClassFactory();

    // IUnknown methods
    ULONG AddRef();
    ULONG Release();
    HRESULT QueryInterface( REFIID riid, void** ppv );

    // IClassFactory methods
    HRESULT CreateInstance( IUnknown* pUnkOuter, REFIID riid, void** ppv );
    HRESULT LockServer( BOOL fLock );

protected:
    ULONG m_uRefCount;
};
```

The constructor, destructor, and IUnknown methods are done just like the earlier sample, so the only new things are the IClassFactory methods. LockServer() is, as you might expect, rather simple:

```
HRESULT CSimpleMsgBoxClassFactory::LockServer ( BOOL fLock )
{
    fLock ? g_uDllLockCount++ : g_uDllLockCount--;
    return S_OK;
}
```

Now for the interesting part, CreateInstance(). Recall that this method is responsible for creating new CSimpleMsgBoxImpl objects. Let's take a closer look at the prototype and parameters:

```
HRESULT CSimpleMsgBoxClassFactory::CreateInstance ( IUnknown* pUnkOuter,
                                                    REFIID    riid,
                                                    void**    ppv );
```

pUnkOuter is only used when this new object is being aggregated, and points to the "outer" COM object, that is, the object that will contain the new object. Aggregation is way beyond the scope of this article, and our sample object will not support aggregation.

riid and ppv are used just as in QueryInterface() - they are the IID of the interface the client is requesting, and a pointer-sized buffer to store the interface pointer.

Here's the CreateInstance() implementation. It starts with some parameter validation and initialization.

```
HRESULT CSimpleMsgBoxClassFactory::CreateInstance ( IUnknown* pUnkOuter,
                                                    REFIID    riid,
                                                    void**    ppv )
{
    // We don't support aggregation, so pUnkOuter must be NULL.
    if ( NULL != pUnkOuter )
```

```
        return CLASS_E_NOAGGREGATION;

    // Check that ppv really points to a void*.
    if ( IsBadWritePtr ( ppv, sizeof(void*) ))
        return E_POINTER;

    *ppv = NULL;
```

We've checked that the parameters are valid, so now we can create a new object.

```
CSimpleMsgBoxImpl* pMsgbox;

    // Create a new COM object!
    pMsgbox = new CSimpleMsgBoxImpl;

    if ( NULL == pMsgbox )
        return E_OUTOFMEMORY;
```

Finally, we QI() the new object for the interface that the client is requesting. If the QI() fails, then the object is unusable, so we delete it.

```
HRESULT hrRet;

    // QI the object for the interface the client is requesting.
    hrRet = pMsgbox->QueryInterface ( riid, ppv );

    // If the QI failed, delete the COM object since the client isn't able
    // to use it (the client doesn't have any interface pointers on the
   //  object).
    if ( FAILED(hrRet) )
        delete pMsgbox;

    return hrRet;
}
```

DllGetClassObject()

Let's take a closer look at the internals of DllGetClassObject(). Its prototype is:

```
HRESULT DllGetClassObject ( REFCLSID rclsid, REFIID riid, void** ppv );
```

rclsid is the CLSID of the coclass the client wants. The function must return the class factory for that coclass.

riid and ppv are, again, like the parameters to QI(). In this case, riid is the IID of the interface that the COM library is requesting on the class factory object. This is usually IID_IClassFactory.

Since DllGetClassObject() creates a new COM object (the class factory), the code looks rather similar to IClassFactory::CreateInstance(). We start off with some validation and initialization.

```
HRESULT DllGetClassObject ( REFCLSID rclsid, REFIID riid, void** ppv )
{
    // Check that the client is asking for the CSimpleMsgBoxImpl factory.
    if ( !InlineIsEqualGUID ( rclsid, __uuidof(CSimpleMsgBoxImpl) ))
        return CLASS_E_CLASSNOTAVAILABLE;

    // Check that ppv really points to a void*.
    if ( IsBadWritePtr ( ppv, sizeof(void*) ))
        return E_POINTER;

    *ppv = NULL;
```

The first if statement checks the rclsid parameter. Our server only contains one coclass, so rclsid must be the CLSID of our CSimpleMsgBoxImpl class. The __uuidof operator retrieves the GUID assigned to CSimpleMsgBoxImpl earlier with the __declspec(uuid()) declaration. InlineIsEqualGUID() is an inline function that checks if two GUIDs are equal.

The next step is to create a class factory object.

```
CSimpleMsgBoxClassFactory* pFactory;

    // Construct a new class factory object.
    pFactory = new CSimpleMsgBoxClassFactory;

    if ( NULL == pFactory )
        return E_OUTOFMEMORY;
```

Here's where things differ a bit from `CreateInstance()`. Back in `CreateInstance()`, we just called `QI()`, and if it failed, we deleted the COM object. Here is a different way of doing things.

We can consider ourselves to be a client of the COM object we just created, so we call `AddRef()` on it to make its reference count 1. We then call `QI()`. If `QI()` is successful, it will `AddRef()` the object again, making the reference count 2. If `QI()` fails, the reference count will remain 1.

After the `QI()` call, we're done using the class factory object, so we call `Release()` on it. If the `QI()` failed, the object will delete itself (because the reference count will be 0), so the end result is the same.

```
    // AddRef() the factory since we're using it.
    pFactory->AddRef();

HRESULT hrRet;

    // QI() the factory for the interface the client wants.
    hrRet = pFactory->QueryInterface ( riid, ppv );

    // We're done with the factory, so Release() it.
    pFactory->Release();

    return hrRet;
}
```

QueryInterface() revisited

I showed a `QI()` implementation earlier, but it's worth seeing the class factory's `QI()` since it is a realistic example, in that the COM object implements more than just `IUnknown`. First we validate the `ppv` buffer and initialize it.

```
HRESULT CSimpleMsgBoxClassFactory::QueryInterface( REFIID riid, void** ppv )
{
HRESULT hrRet = S_OK;

    // Check that ppv really points to a void*.
    if ( IsBadWritePtr ( ppv, sizeof(void*) ))
        return E_POINTER;

    // Standard QI initialization - set *ppv to NULL.
    *ppv = NULL;
```

Next we check `riid` and see if it's one of the interfaces the class factory implements: `IUnknown` or `IClassFactory`.

```
    // If the client is requesting an interface we support, set *ppv.
    if ( InlineIsEqualGUID ( riid, IID_IUnknown ))
        {
        *ppv = (IUnknown*) this;
        }
    else if ( InlineIsEqualGUID ( riid, IID_IClassFactory ))
        {
        *ppv = (IClassFactory*) this;
        }
    else
        {
        hrRet = E_NOINTERFACE;
        }
```

Finally, if `riid` was a supported interface, we call `AddRef()` on the interface pointer, then return.

```
    // If we're returning an interface pointer, AddRef() it.
    if ( S_OK == hrRet )
        {
        ((IUnknown*) *ppv)->AddRef();
        }

    return hrRet;
}
```

The ISimpleMsgBox implementation

Last but not least, we have the code for the one and only method of `ISimpleMsgBox`, `DoSimpleMsgBox()`. We first use the Microsoft extension class `_bstr_t` to convert `bsMessageText` to a `TCHAR` string.

```
HRESULT CSimpleMsgBoxImpl::DoSimpleMsgBox ( HWND hwndParent,
                                            BSTR bsMessageText )
{
_bstr_t bsMsg = bsMessageText;
LPCTSTR szMsg = (TCHAR*) bsMsg;         // Use _bstr_t to convert the
                                        // string to ANSI if necessary.
```

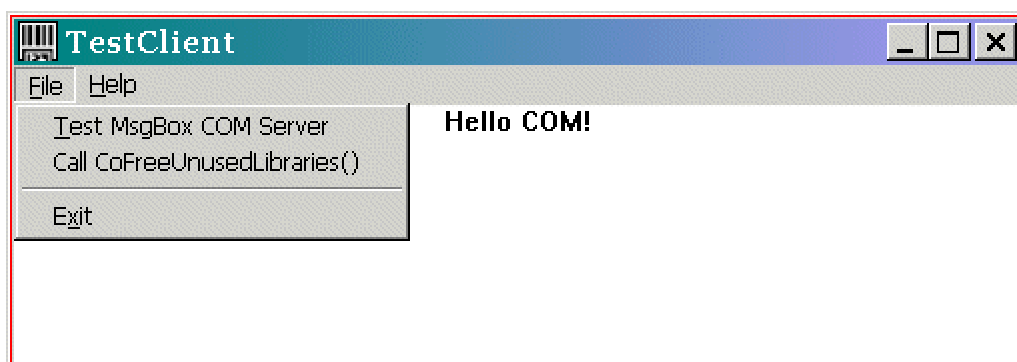
After we do the conversion, we show the message box, and then return.

```
    MessageBox ( hwndParent, szMsg, _T("Simple Message Box"), MB_OK );
    return S_OK;
}
```

A Client to Use Our Server

So now that we've got this super-spiffy COM server all done, how do we use it? Our interface is a *custom interface*, which means it can only be used by a C or C++ client. (If our coclass also implemented `IDispatch`, then we could write a client in practically anything - Visual Basic, Windows Scripting Host, a web page, PerlScript, etc. But that discussion is best left for another article.) I've provided a simple app that uses `ISimpleMsgBox`.

The app based on the Hello World sample built by the Win32 Application AppWizard. The `File` menu contains two commands for testing the server:



The `Test MsgBox COM Server` command creates a `CSimpleMsgBoxImpl` object and calls `DoSimpleMsgBox()`. Since this is a simple method, the code isn't very long. We first create a COM object with `CoCreateInstance()`.

```
void DoMsgBoxTest(HWND hMainWnd)
{
ISimpleMsgBox* pIMsgBox;
HRESULT hr;

hr = CoCreateInstance ( __uuidof(CSimpleMsgBoxImpl), // CLSID of coclass
                        NULL,                        // no aggregation
                        CLSCTX_INPROC_SERVER,        // use only in-proc
                                                     // servers
                        __uuidof(ISimpleMsgBox),     // IID of the interface
                                                     // we want
                        (void**) &pIMsgBox );        // buffer to hold the
```

```
                                                        // interface pointer

      if ( FAILED(hr) )
          return;
```

Then we call DoSimpleMsgBox() and release our interface.

```
      pIMsgBox->DoSimpleMsgBox ( hMainWnd, _bstr_t("Hello COM!") );
      pIMsgBox->Release();
  }
```

That's all there is to it. There are many TRACE statements throughout the code, so if you run the test app in the debugger, you can see where each method in the server is being called.

The other File menu command calls the CoFreeUnusedLibraries() API so you can see the server's DllCanUnloadNow() function in action.

Other Details

COM macros

There are several macros used in COM code that hide implementation details and allow the same declarations to be used by C and C++ clients. I haven't used the macros in this article, but the sample project does use them, so you need to understand what they mean. Here's the proper declaration of ISimpleMsgBox:

```
struct ISimpleMsgBox : public IUnknown
{
    // IUnknown methods
    STDMETHOD_(ULONG, AddRef)() PURE;
    STDMETHOD_(ULONG, Release)() PURE;
    STDMETHOD(QueryInterface)(REFIID riid, void** ppv) PURE;

    // ISimpleMsgBox methods
    STDMETHOD(DoSimpleMsgBox)(HWND hwndParent, BSTR bsMessageText) PURE;
};
```

STDMETHOD() includes the virtual keyword, a return type of HRESULT, and the __stdcall calling convention. STDMETHOD_() is the same, except you can specify a different return type. PURE expands to "=0" in C++ to make the function a pure virtual function.

STDMETHOD() and STDMETHOD_() have corresponding macros used in the implementation of methods - STDMETHODIMP and STDMETHODIMP_(). For example, here's the implementation of DoSimpleMsgBox():

```
STDMETHODIMP CSimpleMsgBoxImpl::DoSimpleMsgBox ( HWND hwndParent,
                                                 BSTR bsMessageText )
{
  ...
}
```

Finally, the standard exported functions are declared with the STDAPI macro, such as:

```
STDAPI DllRegisterServer()
```

STDAPI includes the return type and calling convention. One downside to using STDAPI is that you can't use __declspec(dllexport) with it, because of how STDAPI expands. You instead have to export the function using a .DEF file.

Server registration and unregistration

The server implements the DllRegisterServer() and DllUnregisterServer() functions that I mentioned earlier. Their job is to create and delete the registry entries that tell COM about our server. The code is all boring registry manipulation, so I won't repeat it here, but here's a list of the registry entries created by DllRegisterServer():

Key name	Values in the key
HKEY_CLASSES_ROOT	
CLSID	
{7D51904E-1645-4a8c-BDE0-0F4A44FC38C4}	Default="SimpleMsgBox class"
InProcServer32	Default=[path to DLL]; ThreadingModel="Apartment"

Notes about the sample code

The included sample code contains the source for both the COM server and the test client app. There is a workspace file, SimpleComSvr.dsw, which you can load to work on both the server and client app at the same time. At the same level as the workspace are two header files that are used by both projects. Each project is then in its own subdirectory.

The common header files are:

- ISimpleMsgBox.h - The ISimpleMsgBox definition.
- SimpleMsgBoxComDef.h - Contains the __declspec(uuid()) declarations. These declarations are in a separate file because the client needs the GUID of CSimpleMsgBoxImpl, but not its definition. Moving the GUID to a separate file lets the client have access to the GUID without being dependent on the internal structure of CSimpleMsgBoxImpl. It's the *interface*, ISimpleMsgBox, that's important to the client.

As mentioned earlier, you need a .DEF file to export the four standard exported functions from the server. The sample project's .DEF file looks like this:

```
EXPORTS
    DllRegisterServer   PRIVATE
    DllUnregisterServer PRIVATE
    DllGetClassObject   PRIVATE
    DllCanUnloadNow     PRIVATE
```

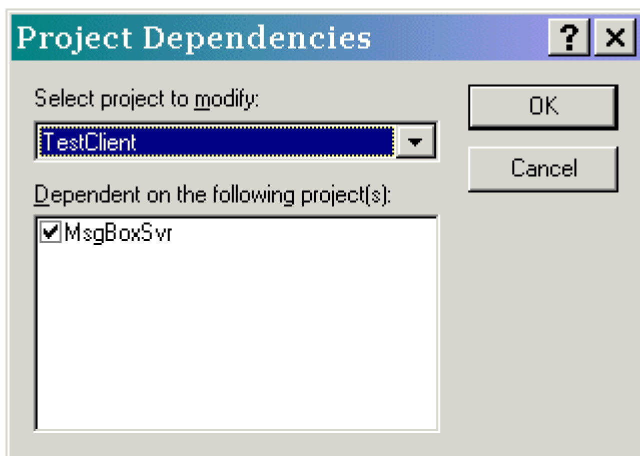
Each line contains the name of the function and the PRIVATE keyword. This keyword means the function is exported, but not included in the import lib. This means that clients can't call the functions directly from code, even if they link with the import lib. This is a required step, and the linker will complain if you leave out the PRIVATE keywords.

Setting breakpoints in the server

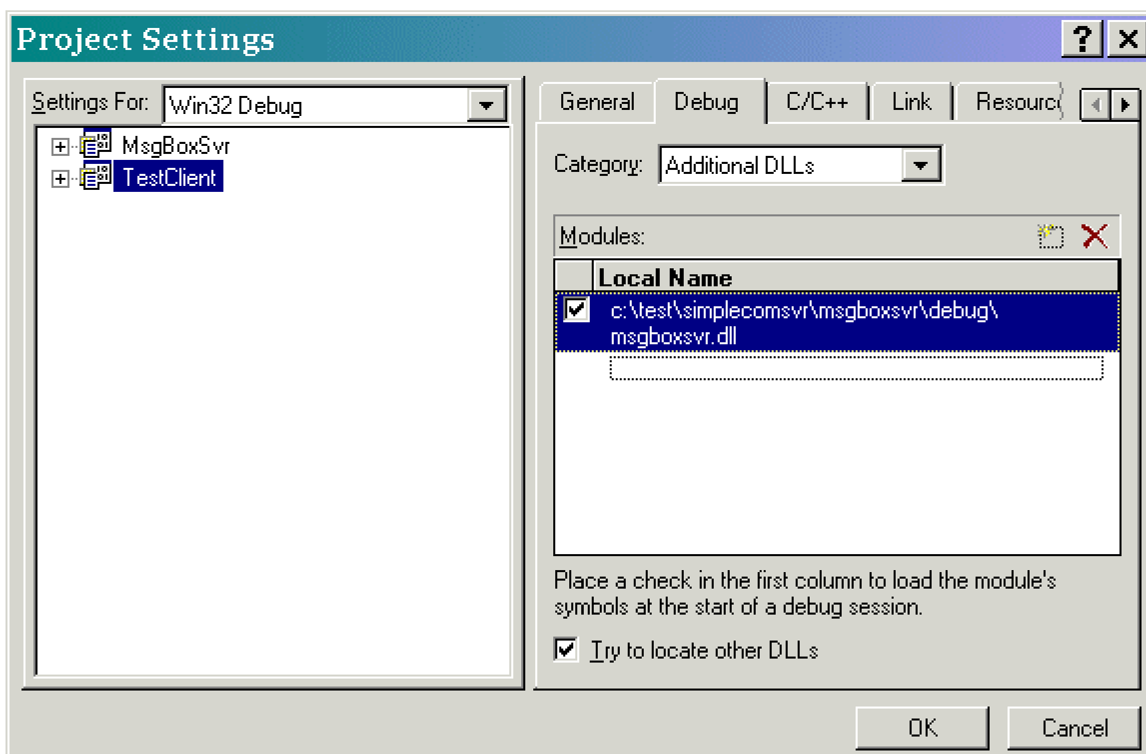
If you want to set breakpoints in the server code, you have two ways of doing it. The first way is to set the server project (MsgBoxSvr) as the active project and then begin debugging. MSVC will ask you for the executable file to run for the debug session. Enter the full path to the test client, which you must already have built.

The other way is to make the client project (TestClient) the active project, and configure the project dependencies so that the server project is a dependency of the client project. That way, if you change code in the server, it will be rebuilt automatically when you build the client project. The last detail is to tell MSVC to load the server's symbols when you begin debugging the client.

The Project Dependencies dialog should look like this:



To load the server's symbols, open the TestClient project settings, go to the Debug tab, and select Additional DLLs in the Category combo box. Click in the list box to add a new entry, and then enter the full path to the server DLL. Here's an example:



The path to the DLL will, naturally, be different depending on where you extract the source code.

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found here

About the Author

Michael Dunn

Michael lives in sunny Mountain View, California. He started programming with an Apple //e in 4th grade, graduated from UCLA with a math degree in 1994, and immediately landed a job as a QA engineer at Symantec, working on the Norton AntiVirus team. He pretty much taught himself Windows and MFC programming, and in 1999 he designed and coded a new interface for Norton AntiVirus 2000. Mike has been a a developer at Napster and at his own lil' startup, Zabersoft, a development company he co-founded with offices in Los Angeles and Odense, Denmark. Mike is now a senior engineer at VMware.

He also enjoys his hobbies of playing pinball, bike riding, photography, and the occasional 360 or MAME game (current favorite: *Space Invaders Extreme*). He

Member

would get his own snooker table too if they weren't so darn big! He is also sad that he's forgotten the languages he's studied: French, Mandarin Chinese, and Japanese.

Mike was a VC MVP from 2005 to 2009.

Occupation:	Software Developer (Senior)
Company:	VMware
Location:	United States

Discussions and Feedback

161 messages have been posted for this article. Visit **http://www.codeproject.com/KB/COM/comintro2.aspx** to post and view comments on this article, or click **here** to get a print view with messages.

PermaLink | Privacy | Terms of Use
Last Updated: 3 Apr 2001
Editor: Rob Manderson

Copyright 2001 by Michael Dunn
Everything else Copyright © CodeProject, 1999-2010
Web12 | Advertise on the Code Project