COM General Technical Articles

The COM Programmer's Cookbook

Crispin Goswell
Microsoft Office Product Unit

Spring 1995
Revised: September 13, 1995

Abstract

This cookbook shows you how to create Microsoft® OLE Component Object Model (COM) objects and use them effectively. The examples are mostly in C, as this shows most clearly what is actually being done. Some programmers will prefer to use C++ to implement their objects. Kraig Brockschmidt's book *Inside OLE* (2nd edition) (MSDN Library, Books) covers the concepts of COM and basic usage from the C++ programmer's perspective. Readers interested in gaining a better understanding of what COM is, as well as the motivations behind its design and philosophy, should read the first two chapters of the *Component Object Model Specification* (MSDN Library, Specifications). Chapter 1 is a brief introduction, and Chapter 2 provides a thorough overview. This cookbook builds on the information found in *Inside OLE* and the COM specification by showing some good ways to implement COM objects.

The Component Object Model

Creating a Microsoft® OLE Component Object Model (COM) object simply requires you to expose one or more interfaces that follow the COM rules. There are many other ways to structure software components. What makes COM interesting is that it is a widely accepted interoperability standard. When you use COM to connect pieces of software, deviations from the standard just cause interoperability problems. I mention this because there are many perfectly reasonable variants and alternatives to COM. The problem with these is that they do not interoperate with other COM objects. This cookbook focuses solely on what you can do with COM implementation within the constraints of the COM standard.

A COM class may be registered in the system registry in order to become a COM server. This is an optional step: It increases the usefulness of a COM object but is not required. For COM objects that are part of some larger whole, it is sometimes not appropriate.

The creation of object instances and the use of interfaces obtained from those instances are often done by separate pieces of code. This separation allows the interface user to be attached to different classes at different times, and is an important feature of COM. Most existing software assumes great knowledge about the implementation details of the code that it calls. By deliberately avoiding such coupling, COM encourages a style of programming in which different pieces of code can be truly independent of one another. Rather than assuming knowledge about the implementation of some called code, a COM client assumes a *service* from an interface or collection of interfaces. Different implementations can provide that service. COM interfaces provide a standard means by which a client of an object can perform an initial negotiation for a particular service and then assume some behavior from that service that conforms to a known contract.

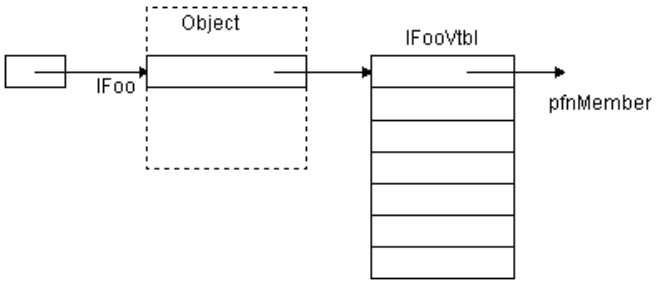
COM Interfaces

The separation between service user and implementation is done by indirect function calls. A COM interface is nothing more than a named table of function pointers (*methods*), each of which has documented behavior. The behavior is documented in terms of the interface function's parameters and a *model* of the state within the object instance. The description of the model within the instance will say no more than is required to make the behavior of the other methods in the interface understandable. The table of functions is referred to as a *vtable*. Here is an example interface expressed in C:

```
typedef struct
{
    struct IFooVtbl *lpVtbl;
} IFoo;

typedef struct
{
```

```
    // IUnknown methods
    HRESULT     (*QueryInterface) (IFoo *, REFIID, void **);
    ULONG       (*AddRef)         (IFoo *);
    ULONG       (*Release)        (IFoo *);
    // IFoo methods
    HRESULT     (*SetValue)     (IFoo *, int);
    HRESULT     (*GetValue)     (IFoo *,int *);
} IFooVtbl;
```

An interface is actually a pointer to a vtable. The vtables are usually shared by multiple instances, so the methods need a different pointer to be able to find the object that the interface is attached to. This is the interface pointer, and the vtable pointer is the only thing that is accessible from it by clients of the interface. By design, this arrangement matches the virtual method-calling convention of C++ classes, so a COM interface is binary-compatible with a C++ abstract class.



An interface can be declared in a way that is usable from C or C++ using the COM macros. Here is IFOO.H:

```
#include <objbase.h>

#undef INTERFACE
#define INTERFACE IFoo

DECLARE_INTERFACE_ (IFoo, IUnknown)
{
    // IUnknown methods
    STDMETHOD (QueryInterface)    (THIS_  REFIID, void **) PURE;
    STDMETHOD_(ULONG, AddRef)     (THIS) PURE;
    STDMETHOD_(ULONG, Release)    (THIS) PURE;

    STDMETHOD (SetValue)          (THIS_  int) PURE;
    STDMETHOD (GetValue)          (THIS_  int *) PURE;
};

// {A46C12C0-4E88-11ce-A6F1-00AA0037DEFB}
DEFINE_GUID(IID_IFoo, 0xa46c12c0, 0x4e88, 0x11ce, 0xa6, 0xf1, 0x0, 0xaa, 0x0, 0x37, 0xde, 0xfb);
```

This declaration expands into the C **typedefs** described earlier. The last two lines were created using GUIDGEN. This creates a unique 16-byte number for the *interface ID* (IID_IFoo here) and can copy it to the Clipboard for pasting into source code. An IID (which appears as the parameter type **REFIID** in the **QueryInterface** method above) is a means of identifying a particular interface for the purposes of run-time lookup.

The interface can also be described in interface definition language (IDL), which is input to a tool, Microsoft IDL (MIDL), which can produce a header file from a specification such as the following (IFOO.IDL):

```
[ object, uuid(A46C12C0-4E88-11ce-A6F1-00AA0037DEFB) ]
interface IFoo : IUnknown
{
    HRESULT SetValue ([in] int);
    HRESULT GetValue ([out] int *);
};
```

MIDL is also able to generate COM remoting code (remote procedure calls [RPC]) from this description. This allows one process to call interfaces across process or machine boundaries.

The COM Interface Rules

A full description of the COM interface rules can be found in the COM specification or in the technical article "The Rules of the Component Object Model" in the Microsoft Development (MSDN) Library and in the COM Resource Kit.

Here is a summary of the rules:

- The first three methods in a COM interface are required to be the same for all COM interfaces. These are the methods of the **IUnknown** interface: **QueryInterface**, **AddRef**, and **Release**. C++ programmers would say that the interface must *inherit* from **IUnknown**.
- The **QueryInterface** method takes an interface ID (an IID) and returns the requested COM interface on the same object. The set of interface IDs accessible via **QueryInterface** is the same from every interface on the object.
- **QueryInterface** must have stable behavior: If it succeeds once for a particular interface on a particular instance, it should always succeed. If it fails once for a particular interface, it should always fail. If it succeeds, it should return NOERROR; otherwise, it should normally return E_NOINTERFACE and, importantly, zero its out parameter.
- The interface returned by **QueryInterface** in response to a request for **IID_IUnknown** (the interface ID for **IUnknown**) should have the same pointer value at all times and from all interfaces on that instance. This is not required of any other interface. This pointer value is the instance's *identity*. The identity of an object is the way in which one compares two objects to see if they are actually the same one. This can be found only by comparing the **IUnknown** interface pointers. You can find the identity of the object behind any interface by first doing a **QueryInterface** for **IUnknown** and then comparing the pointers.
- Each *interface* has a nominal reference count associated with it. When a piece of code *receives* an interface, it should assume the reference count is notionally 1 unless it has other knowledge. **AddRef** increments the count, **Release** decrements it. When the client has made one more call to **Release** than to **AddRef** (meaning that the nominal count is zero), then it must assume that the interface pointer is invalid. The provider of the interface may choose to invalidate the interface pointer at that point (or not), but not before. Some implementations have more relaxed behavior (for example, sharing reference counts for all interfaces and/or only invalidating interface pointers when the object is freed), but such behavior is not required, and should certainly not be depended on. Although **AddRef** and **Release** return a ULONG, the value returned is not required to have any meaning. Client code should not depend in any way on the value returned. It exists solely for debugging code to use, typically to return the new object reference count.

The reference-counting rules are very simple. When a client calls a member function and passes one or more interface pointers:

- The callee should **AddRef** its interface parameters if it wants them to stay around after it has gone.
- The caller should **Release** the callee's returned interfaces when it is done with them.

A straightforward consequence of the above rules is that **QueryInterface** does an implicit **AddRef** on the returned interface, as do the application programming interfaces (APIs) that create objects, such as **CoCreateInstance**.

Reference-Counting

In spite of the straightforward rules, getting the reference-counting right requires some care. Programmers coming from languages that do garbage collection have a similar problem with remembering to free heap blocks. From the reference-counting viewpoint, an interface pointer is a heap block that multiple people are using, so they all have to stake a claim to it, and all have to free it when they're done.

Bug detection

What typically goes wrong is that one user forgets to stake that claim (with **AddRef**), or forgets to **Release** it when he or she is done. When that happens there's no obvious way to find out who was responsible.

A surplus **AddRef** can be detected late by watching the return value of **Release** (debug only) and Asserting that it reaches zero when expected. This can be done when an object is releasing an aggregated object (see below) during its own release, or a contained object that it knows should not be externally referenced. I use a macro, **ReleaseLast**, which asserts that the **Release** returns zero.

A surplus **Release** can be detected by zeroing out the vtable pointer of the **Released** interface or object when the reference count reaches zero. Obviously this works better with interface-specific reference counts than when all the object's interfaces are sharing a reference count. I just zero memory when I free it. It's also generally a good idea to zero interface pointers when you release them, as in most circumstances you do not know whether they will be valid afterward or not. This will often detect surplus **Releases** when they happen rather than later.

Bug avoidance

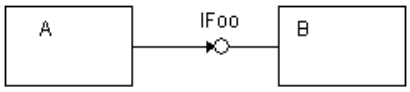
Like a lot of programming issues, avoidance is mostly a matter of good programming practices and conventions. Many interface uses follow the pattern:

```
QueryInterface () // (or some other function or method that returns
                     an interface pointer)
    // ...some method calls ...
    Release ().
```

A good practice is to keep these lines as close together as possible. I tend to use extra indentation for a zone of code where there is a temporary **AddRef** outstanding. The same advice can be applied to any kind of resource management.

Boxology

COM has its own visual notation, which looks like this:



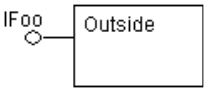
These diagrams represent the connectivity between instances of COM objects in an active system. This is quite different from class hierarchies in C++, or flowcharts in procedural programming. The square boxes represent object instances—specifically the *data* encapsulated by an object and usually a label identifying the class of the code associated with it.

The small circles represent individual interfaces exposed by an object, of which there may be many. An arrow represents a *use* of an interface, of which there may also be many. In this diagram, object A uses the **IFoo** interface on object B.

I was amused to learn recently that some people refer to the interface pictures as "lollipops."

Building a COM Component

So, putting all this together, here is a simple COM class, OUTSIDE.C:



This one implements **IUnknown** and **IFoo; IFoo** has members to set and get a value. It's not a very interesting class. I should point out that it is, intentionally, mostly COM mechanism. A reasonable COM class has an overhead of about a page of code, but this is roughly constant.

Each member function uses the **IMPL** macro to get from the interface pointer to the **COutside** instance data. We call the result "this" because it matches closely what C++ does for you. **FooQueryInterface** returns a pointer to the **IFoo** interface structure in response to **IID_IUnknown** or **IID_IFoo**. If the IID is not either of those, it return E_NOINTERFACE and zeros its out parameter, *ppv* (as required by the interface specification). Note that we are using the fact that all interfaces are "prefixed" with the **IUnknown** methods to implement **IUnknown** for free, thus these two interfaces are also sharing the reference count. (It is important to remember that two or more interfaces sharing a single reference count is an implementation detail of the object. Users of COM objects, however, must always assume interface counting is per-interface.)

The **typedef** ... **COutside** describes the layout of the instance data for this class of object, and the **CreateOutside** function allocates this structure and fills it in. The **QueryInterface** call at the end allows the created object to return the requested interface. The **Release** afterwards causes the object to be deallocated again if the interface requested was not available.

The **CreateOutside** function has a signature that makes it suitable for use in a generic class factory. The class factory may be associated with a CLSID (Class ID) that eventually appears in the system registry. A class factory is an object that exists to create instances of other objects. It exists to separate the process of finding a class implementation from the process of creating instances. The CLSID is a 16-byte number that uniquely identifies the class (and thus its class factory). These are described later in more detail.

The **CreateOutside** function is the only "extern" object here: We declare other functions static so that we can use the file as

the unit of scoping. Static functions do not interfere with other code, so we can be a little more relaxed about Hungarian notation than would otherwise be necessary in C. (Note that in the code samples that follow, I've highlighted sections deserving special attention with rows of dashes.)

```
#include "util.h"
#include "ifoo.h"

typedef struct
{
    IFoo ifoo;
    int cRef;
    int value;
} COutside;

static HRESULT FooQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
//--------------------------------------------------------------------
    COutside *this = IMPL (COutside, ifoo, pfoo);
\\--------------------------------------------------------------------
    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
    else
    {
//--------------------------------------------------------------------
        *ppv = 0;
\\--------------------------------------------------------------------
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG FooAddRef (IFoo *pfoo)
{
//--------------------------------------------------------------------
    COutside *this = IMPL (COutside, ifoo, pfoo);
\\--------------------------------------------------------------------
    return ++this->cRef;
}

static ULONG FooRelease (IFoo *pfoo)
{
//--------------------------------------------------------------------
    COutside *this = IMPL (COutside, ifoo, pfoo);
\\--------------------------------------------------------------------
    if (--this->cRef == 0)
    {
//--------------------------------------------------------------------
        --vcObjects;
\\--------------------------------------------------------------------

        Free (this);
        return 0;
    }
    return this->cRef;
}

static HRESULT FooSetValue (IFoo *pfoo, int value)
{
//--------------------------------------------------------------------
    COutside *this = IMPL (COutside, ifoo, pfoo);
\\--------------------------------------------------------------------
    this->value = value;
    return NOERROR;
}

static HRESULT FooGetValue (IFoo *pfoo, int *pValue)
{
//--------------------------------------------------------------------
    COutside *this = IMPL (COutside, ifoo, pfoo);
\\--------------------------------------------------------------------
```

```
    if (!pValue)
        return E_POINTER;
    *pValue = this->value;
    return NOERROR;
}

//--------------------------------------------------------------------
static IFooVtbl vtblFoo =
{
    FooQueryInterface, FooAddRef, FooRelease,
    FooSetValue,
    FooGetValue
};
\\--------------------------------------------------------------------

HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;
    if (hr = Alloc (sizeof (COutside), &this))
        return hr;
//--------------------------------------------------------------------
    this->ifoo.lpVtbl = &vtblFoo;
    this->cRef = 1;
    this->value = 0;
    ++vcObjects;
    hr = QueryInterface (&this->ifoo, riid, ppv);

    Release (&this->ifoo);
\\--------------------------------------------------------------------
    return hr;
}
```

In the creation function above, we initialize the reference count to 1 so that should the initial **QueryInterface** fail, the subsequent **Release** will cleanly deallocate the object.

For comparison, here's the same class in C++:

```
#include "ifoo.h"

extern "C" int vcObjects;

struct COutside : IFoo
{
//--------------------------------------------------------------------
//    IUnknown methods
    HRESULT __stdcall QueryInterface (REFIID riid, void **ppv);
    ULONG __stdcall AddRef (void);

    ULONG __stdcall Release (void);

//    IFoo methods
    HRESULT __stdcall SetValue (int value);
    HRESULT __stdcall GetValue (int *pvalue);

    COutside (void);
\\--------------------------------------------------------------------

    int m_cRef;
    int m_value;
};

HRESULT COutside::QueryInterface (REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown || riid == IID_IFoo)
        *ppv = (IFoo *) this;
    else
```

```
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }


    ((IUnknown *)*ppv)->AddRef ();
    return NOERROR;
}

ULONG COutside::AddRef (void)
{
    return ++ m_cRef;
}

ULONG COutside::Release (void)
{
    if (--m_cRef == 0)
    {
        --vcObjects;

        delete this;
        return 0;
    }
    return m_cRef;
}

HRESULT COutside::SetValue (int v)
{
    m_value = v;
    return NOERROR;
}

HRESULT COutside::GetValue (int *pv)
{
    if (!pv)
        return E_POINTER;
    *pv = m_value;
    return NOERROR;
}

inline COutside::COutside (void)
{
    m_cRef = 1;
    m_value = 0;
    ++vcObjects;
}

extern "C" HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *pout;
    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;
    if (!(pout = new COutside))
        return E_OUTOFMEMORY;
    HRESULT hr = pout->QueryInterface (riid, ppv);

    pout->Release ();
    return hr;
}
```

This example looks smaller and cleaner than the equivalent C code. There are several things to notice:

- The class declaration, **COutside**, is in the C++ source file, not a header file. This is because the implementation is private and COM does not encourage classes to share implementation with inheritance.
- We have to list the call signature of every member function that we implement in the class declaration (which means all of them) and again when we write the code.
- We don't have to initialize the vtable, as we do in C.
- We don't have to set up or use the **this** pointers in this simple example.

- The class name, **COutside**, is joined with the member names and the member names are "extern", so implementation class names have to be unique across the whole DLL/EXE being implemented and across any libraries it includes. In C, the creation function is the only extern symbol, and it is used only in one place, so giving it a long name is not a real burden.

The generated executable code for the C and C++ examples are almost identical.

Here is some sample code that uses the class:

```
#include "util.h"
#include "ifoo.h"

// {8836A5A0-4E8A-11ce-A6F1-00AA0037DEFB}
DEFINE_GUID(CLSID_Outside, 0x8836a5a0, 0x4e8a, 0x11ce, 0xa6, 0xf1, 0x0, 0xaa, 0x0, 0x37, 0xde, 0xfb);

void SomeFunction ()
{
    IFoo *pfoo;
    HRESULT hr;
    int result;

    hr = CoCreateInstance (&CLSID_Outside, NULL, CLSCTX_SERVER, &IID_IFoo,
                           &pfoo);
        pfoo->lpVtbl->SetValue (pfoo, 42);
        // ....
        pfoo->lpVtbl->GetValue (pfoo, &result);
        // ....
        Release (pfoo);
}
```

CoCreateInstance is a COM API which takes the CLSID for a class, finds its class factory, and calls its **CreateInstance** member. The requested interface for the newly created object is returned in **Pfoo**. The NULL parameter is described below in the section on aggregation. The *CLSCTX_SERVER* parameter can be used to constrain the contexts in which the server may run. See the OLE/COM documentation for further details. Following that and for illustrative purposes, the above code calls the **SetValue** and **GetValue** members of the **IFoo** interface on the new object and then releases the interface. Because there are no other references to the object in this case, this will cause the new object instance to be freed.

Here is my UTIL.H:

```
#include <objbase.h>

#define VTABLE(ob, member) (*((ob)->lpVtbl->member))

#define IUNK_VTABLE_OF(x) ((IUnknownVtbl *)((x)->lpVtbl))

#define QueryInterface(pif, iid, pintf) \
    (IUNK_VTABLE_OF(pif)->QueryInterface((IUnknown *)(pif),
                                         iid, (void **)(pintf)))

#define AddRef(pif) \
    (IUNK_VTABLE_OF(pif)->AddRef((IUnknown *)(pif)))

#define Release(pif) \
....(IUNK_VTABLE_OF(pif)->Release((IUnknown *)(pif)))

// from stddef.h
#ifndef offsetof
#define offsetof(s,m)  (size_t)&(((s *)0)->m)
#endif

#define IMPL(class, member, pointer) \
    (&((class *)0)->member == pointer, ((class *) (((long) pointer) - offsetof
                                                   (class, member))))

extern HRESULT Alloc (size_t, void **ppv);
extern HRESULT Free (void *pv);

#define QITYPE HRESULT (*)(void *, REFIID, void **)
#define ARTYPE ULONG (*)(void *)
```

```
#define RLTYPE ULONG (*)(void *)

extern int vcObjects;
```

QueryInterface, **AddRef**, and **Release** are defined to enable us to leave out the **punk->lpVtbl->** at the front of every method call. The OLE headers contain macros for all members of all defined interfaces that support this abbreviation. They are of the form **IUnknown_QueryInterface**, **IClassFactory_CreateInstance**, and so on.

IMPL is a means to get from an *interface* pointer to an *instance* pointer by subtracting the offset of the interface in the object structure. For objects with a single interface that is the first element of the structure, this subtracts zero, which compiles down to nothing. For other interfaces, it compiles to a SUB instruction that subtracts a constant from the interface pointer at the top of each member function. The **IMPL** macro is a "comma expression," which compares the pointer with a NULL pointer of the correct type, and then throws away the result. This allows the macro to be a type-safe cast from one known type to another. Because the result of the comparison is not used, it also compiles to nothing. **Alloc** and **Free** are wrappers for your favorite allocator.

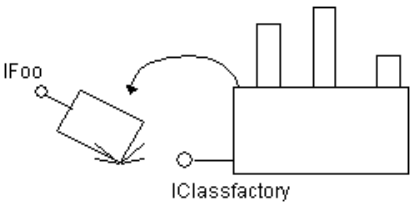
Here is the client example code in C++:

```
#include "ifoo.h"

// {8836A5A0-4E8A-11ce-A6F1-00AA0037DEFB}
DEFINE_GUID(CLSID_Outside, 0x8836a5a0, 0x4e8a, 0x11ce, 0xa6, 0xf1, 0x0, 0xaa, 0x0, 0x37, 0xde, 0xfb);

void SomeFunction ()
{
    IFoo *pfoo;
    HRESULT hr;
    int result;

    hr = CoCreateInstance(CLSID_Outside, NULL, CLSCTX_SERVER, IID_IFoo, &pfoo);
        pfoo->SetValue(42);
        // ....
        pfoo->GetValue(&result);
        // ....
        pfoo->Release();
}
```

Notice that in C++, the calls look a little tidier: The vtable indirection is hidden, and *const &* arguments are used to pass CLSIDs and IIDs around.

Class Factories and COM Servers

A client can always call **CreateOutside** directly to create an instance of the class, but doing so wires in the assumption that a particular implementation will be used, and also that it will be linked with the client code. The COM mechanisms make it possible to be more flexible about locating **Outside**.



COM separates the steps of identifying the code for a class (which involves a lookup in the system registry) from the process of creating an instance of that class, which involves indirectly calling **CreateOutside** in this case.

The code may reside in a dynamic-link library (DLL), the current .DLL or .EXE, another .EXE, or on another machine. When you call **CoCreateInstance**, COM figures out from the registry which will be the nearest at hand.

What about the lookup cost of **CoCreateInstance**? **CoCreateInstance** is actually just a wrapper for **CoGetClassObject** and a call to **IClassFactory::CreateInstance**. If you know you're going to be creating **Outsides** often, but don't want to wire in unnecessary assumptions, you can cache the registry lookup by holding onto the class factory pointer:

```
    hr = CoGetClassObject (&CLSID_Outside, CLSCTX_INPROC_SERVER, NULL,
            &IID_IClassFactory, &pcfFoo);
    if (pcfFoo)
        pcfFoo->LockServer (pcfFoo, TRUE);
    // ....
    while (1)
    {
        pcfFoo->lpVtbl->CreateInstance (pcfOutside, 0, &IID_IFoo, &pfoo);
        // .....
    }
```

It is rare for this to be worth the effort—the class factory lookup is not expensive, though if you're creating **Outside**s very often, it might be significant. Your mileage may vary.

If you keep hold of a class factory, it's important to call **IClassFactory::LockServer**, because a reference count on a class factory is not sufficient to keep a remote server running. This is a rare exception in COM. If you do not call **LockServer**, the server may shut down, after which calls to **CreateInstance** may fail.

The registry entry (OUTSIDE.REG) for **Outside** looks like this:

```
REGEDIT
HKEY_CLASSES_ROOT\CLSID\{8836A5A0-4E8A-11ce-A6F1-00AA0037DEFB} = Outside
HKEY_CLASSES_ROOT\CLSID\{8836A5A0-4E8A-11ce-A6F1-00AA0037DEFB}\InprocServer32 = outside.dll
```

You can enter this data by double-clicking the file in File Manager or by adding keys in REGEDIT. OUTSIDE.DLL is the DLL that contains the in-process server implementation of **Outside** (and its class factory).

Here is OUTDLL.C:

```
#include "util.h"
#include <initguid.h>
#include "ifoo.h"

// {8836A5A0-4E8A-11ce-A6F1-00AA0037DEFB}
DEFINE_GUID(CLSID_Outside, 0x8836a5a0, 0x4e8a, 0x11ce, 0xa6, 0xf1, 0x0, 0xaa, 0x0, 0x37, 0xde, 0xfb);

static IClassFactory *vpcfOutside = 0;
int vcObjects = 0;

HRESULT DllGetClassObject (REFCLSID rclsid, REFIID riid, void **ppv)
{
    *ppv = 0;
    if (IsEqualCLSID (rclsid, &CLSID_Outside))
    {

        if (!vpcfOutside)

        {

            HRESULT hr = CreateClassFactory (&CLSID_Outside, CreateOutside,
                                             &IID_IClassFactory, &vpcfOutside);

                if (hr != NOERROR)

                    return hr;

            }

        return QueryInterface (vpcfOutside, riid, ppv);

    }

    return E_FAIL;
}

HRESULT DllCanUnloadNow ()
{

    return vcObjects == 0 ? S_OK : S_FALSE;
```

```
}
```

How, then, does one build a class factory? Here is an example implementation. This one is not specific to a particular class, so it will serve for most in-process servers that you might want to build:

```
#include "util.h"

typedef struct
{
    IClassFactory icf;
    int cRef;

    HRESULT (*pfnCreate)(IUnknown *, REFIID, void **);
} ClassFactory;

static IClassFactoryVtbl vtblClassFactory;

static HRESULT CFQueryInterface (IClassFactory *pcf, REFIID riid, void **ppv)
{
    ClassFactory *this = IMPL (ClassFactory, icf, pcf);

    if (IsEqualIID (riid, &IID_IUnknown) ||
            IsEqualIID (riid, &IID_IClassFactory))
        *ppv = &this->icf;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IClassFactory *)*ppv);

    return NOERROR;
}

static ULONG CFAddRef (IClassFactory *pcf)
{
    ClassFactory *this = IMPL (ClassFactory, icf, pcf);

    return ++this->cRef;
}

static ULONG CFRelease (IClassFactory *pcf)
{
    ClassFactory *this = IMPL (ClassFactory, icf, pcf);

    if ( --this-> cRef == 0)
    {
        Free (this);
        return 0;
    }

    return this->cRef;
}

static HRESULT CreateInstance (IClassFactory *pcf, IUnknown *punkOuter, REFIID
                               riid, void **ppv)
{
    ClassFactory *this = IMPL (ClassFactory, icf, pcf);

    return (*this->pfnCreate)(punkOuter, riid, ppv);
}

static HRESULT LockServer (IClassFactory *pcf, BOOL flock)
{
    if (flock)
        ++vcObjects;
    else
        --vcObjects;

    return NOERROR;
```

```
}

static IClassFactoryVtbl vtblClassFactory =
{
    CFQueryInterface, CFAddRef, CFRelease,
    CreateInstance,
    LockServer
};

HRESULT CreateClassFactory (REFCLSID rclsid,
    HRESULT (*pfnCreate)(IUnknown *, REFIID, void **),
    REFIID riid, void **ppv)
{
    ClassFactory *this;
    HRESULT hr;

    *ppv = 0;
    if (hr = Alloc (sizeof (ClassFactory), &this))
    return hr;

    this->icf.lpVtbl = &vtblClassFactory;
    this->cRef = 1;

    this->pfnCreate = pfnCreate;

    hr = QueryInterface (&this->icf, riid, ppv);
    Release (&this->icf);

    return hr;
}
```

Note that the class factory does not alter the **vcObjects** global object count, and thus does not prevent the DLL from being unloaded merely by existing.

Simply *becoming* an in-process COM server does not require you to link with any extra libraries or load any DLLs. *Using* a COM object obtained from **CoCreateInstance** requires that the OLE libraries be linked to obtain the **CoCreateInstance** function.

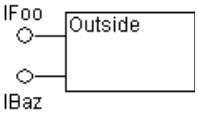
The size of the COM server DLL built above is 3552 bytes; 336 bytes of this is the actual object implementation. The rest is constant DLL overhead.

The story for EXEs is a little different. When the EXE is ready to expose a class factory, it needs to register it using the **CoRegisterClassObject** function (and use **CoRevokeClassObject** when it is ready to shut down). This requires linking with OLE32.LIB and will load the corresponding DLL.

COM Objects with More Than One Interface

C techniques

Building a COM object that supports more than one interface is straightforward. The simplest version is shown below in order to introduce the concepts, and the differences from the previous example are marked with rows of dashes. Following this code is a recommended refinement, which is best understood after looking first at this code.



In the code below, another interface object is added to the instance structure, **FooQueryInterface** gets a clause to expose it, the methods are implemented, the vtable is declared, and the interface is initialized in the creation function. You'll notice that we end up implementing another set of the **IUnknown** methods for **IBaz**. We reuse the implementation in **IFoo** by *delegating* these calls to those functions. This is irksome: Although it would be straightforward to define a macro to make this a one-liner, that doesn't save the increase in EXE size.

```
#include "util.h"
#include "ifoo.h"
```

```
//------------------------------------------------------------------------
#include "ibaz.h"
\\------------------------------------------------------------------------

typedef struct
{
    IFoo ifoo;
//------------------------------------------------------------------------
    IBaz ibaz;
\\------------------------------------------------------------------------
    int cRef;
    int value;
} COutside;

static HRESULT FooQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);

    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
//------------------------------------------------------------------------
    else if (IsEqualIID (riid, &IID_IBaz))
        *ppv = &this->ibaz;
\\------------------------------------------------------------------------
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG FooAddRef (IFoo *pfoo)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);
    return ++this->cRef;
}

static ULONG FooRelease (IFoo *pfoo)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);

    if (--this->cRef == 0)
    {
        --vcObjects;

        Free (this);
        return 0;
    }
    return this->cRef;
}

static HRESULT SetValue (IFoo *pfoo, int value)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);
    this->value = value;
    return NOERROR;
}

static HRESULT GetValue (IFoo *pfoo, int *pValue)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);

    if (!pValue)
        return E_POINTER;
    *pValue = this->value;

    return NOERROR;
}
```

```
static IFooVtbl vtblFoo =
{
    FooQueryInterface, FooAddRef, FooRelease,
    SetValue,
    GetValue
};

//------------------------------------------------------------------
static HRESULT BazQueryInterface (IBaz *pbaz, REFIID riid, void **ppv)
{
    COutside *this = IMPL (COutside, ibaz, pbaz);
    return QueryInterface (&this->ifoo, riid, ppv);
}

static ULONG BazAddRef (IBaz *pbaz)
{
    COutside *this = IMPL (COutside, ibaz, pbaz);
    return AddRef (&this->ifoo);
}

static ULONG BazRelease (IBaz *pbaz)
{
    COutside *this = IMPL (COutside, ibaz, pbaz);
    return Release (&this->ifoo);
}

static HRESULT SquareValue (IBaz *pbaz)
{
    COutside *this = IMPL (COutside, ibaz, pbaz);
    this->value *= this->value;
    return NOERROR;
}

static IBazVtbl vtblBaz =
{
    BazQueryInterface, BazAddRef, BazRelease,
    SquareValue
};
\\------------------------------------------------------------------

HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;

    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

    this->ifoo.lpVtbl = &vtblFoo;
//------------------------------------------------------------------
    this->ibaz.lpVtbl = &vtblBaz;
\\------------------------------------------------------------------
    this->cRef = 1;
    this->value = 0;

    ++vcObjects;

    hr = QueryInterface (&this->ifoo, riid, ppv);

    Release (&this->ifoo);
    return hr;
}
```

The recommended method

One can write **IUnknown** methods that are able to deduce which interface they are called from, and use the same methods in every vtable. This is not expressible in C++, though the adjuster thunks used in multiple inheritance are fairly cheap.

In the example below, the **FindImpl** function replaces the **IMPL** macro in the **IUnknown** members (all other members continue to use **IMPL**). **FindImpl** steps back through the instance structure until it finds the **IUnknown** vtable. Thus, that vtable needs to be the first in the class structure, and they need to be contiguous. This does introduce some extra cost to the **IUnknown** methods. Some benchmarking of common usage will show the degree to which this is significant. Note that the other methods on the interfaces continue to use the **IMPL** macro, and thus do not pay this cost.

Here is the implementation of **FindImpl**:

```
__inline void *FindImpl (void *punkPassed, void *lpVtblFirst)
{
    void **punkVtbl = (void **)punkPassed;
    while (*punkVtbl != lpVtblFirst)
        --punkVtbl;
    return punkVtbl;
}
```

When you have only a couple interfaces, you could check the **lpVtbl** pointer directly, though **FindImpl** is generally cheap enough.

```
    Foo *this = (pfoo->lpVtbl == &vtblFoo) ?
        IMPL (Foo, ifoo, pfoo) :
        IMPL (Foo, ibaz, (IBaz *)pfoo);
```

FindImpl is inlined because the inline expansion is only 10 percent larger than a call to it and involves executing half as many instructions.

Here is the code for OUTSIDE2.C:

```
#include "util.h"
#include "ifoo.h"
#include "ibaz.h"

typedef struct
{
    IFoo ifoo;
    IBaz ibaz;
    int cRef;
    int value;
} COutside;

//------------------------------------------------------------------------
static IFooVtbl vtblFoo;
\\------------------------------------------------------------------------

static HRESULT OutsideQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
//------------------------------------------------------------------------
    COutside *this = FindImpl (pfoo, &vtblFoo);
\\------------------------------------------------------------------------

    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
    else if (IsEqualIID (riid, &IID_IBaz))
        *ppv = &this->ibaz;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG OutsideAddRef (IFoo *pfoo)
{
//------------------------------------------------------------------------
    COutside *this = FindImpl (pfoo, &vtblFoo);
\\------------------------------------------------------------------------
```

```
    return ++this->cRef;
}

static ULONG OutsideRelease (IFoo *pfoo)
{
//--------------------------------------------------------------------
    COutside *this = FindImpl (pfoo, &vtblFoo);
\\--------------------------------------------------------------------

    if (--this->cRef == 0)
    {
        --vcObjects;

        Free (this);
        return 0;
    }
    return this->cRef;
}

static HRESULT SetValue (IFoo *pfoo, int value)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);
    this->value = value;
    return NOERROR;
}

static HRESULT GetValue (IFoo *pfoo, int *pValue)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);

    if (!pValue)
        return E_POINTER;
    *pValue = this->value;

    return NOERROR;
}

static IFooVtbl vtblFoo =
{
//--------------------------------------------------------------------
    (QITYPE) OutsideQueryInterface,
    (ARTYPE) OutsideAddRef,
    (RLTYPE) OutsideRelease,
\\--------------------------------------------------------------------
    SetValue,
    GetValue
};


static HRESULT SquareValue (IBaz *pbaz)
{
    COutside *this = IMPL (COutside, ibaz, pbaz);
    this->value *= this->value;
    return NOERROR;
}

static IBazVtbl vtblBaz =
{
//--------------------------------------------------------------------
    (QITYPE) OutsideQueryInterface,
    (ARTYPE) OutsideAddRef,
    (RLTYPE) OutsideRelease,
\\--------------------------------------------------------------------
    SquareValue
};

HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    *ppv = 0;
```

```
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;

    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

    this->ifoo.lpVtbl = &vtblFoo;
    this->ibaz.lpVtbl = &vtblBaz;
    this->cRef = 1;
    this->value = 0;
    ++vcObjects;
    hr = QueryInterface (&this->ifoo, riid, ppv);

    Release (&this->ifoo);
    return hr;
}
```

Other methods

Another alternative method is to store back pointers with each interface that point to the instance structure. Most of the OLE samples use these back pointers in every method. It might be reasonable to use them only for the **IUnknown** methods, although doing so doubles the space overhead for an interface from four to eight bytes per interface per instance. For situations where the space overhead for the interfaces themselves is too much, there are alternatives in the form of tear-off interfaces. See the section below on optimizations.

Implementing interface-specific reference counts in C

If you fold together the implementation of the **IUnknown** methods, it is not obvious how interface-specific reference-counting can be achieved. Here's one way that it can be done with **FindImpl**. We provide a macro that can generate an interface index number for a given interface pointer:

```
    #define INTERFACE_INDEX(this, pfoo) \
        ((((char *)pfoo) - (char *)this) / sizeof(IUnknown))
```

We then place within the object an array of interface reference counts. The debug code below will assert whether an interface is released one time too many:

```
typedef struct
{
    IFoo ifoo;
    IBaz ibaz;
//------------------------------------------------------------------------
#ifdef DEBUG
    int cRefs[2];
#endif
\\------------------------------------------------------------------------
    int cRef;
    int value;
} COutside;

//------------------------------------------------------------------------
static IFooVtbl vtblFoo;
\\------------------------------------------------------------------------

static ULONG OutsideAddRef (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

//------------------------------------------------------------------------
#ifdef DEBUG
    ++this->cRefs[INTERFACE_INDEX (this, pfoo)];
#endif
\\------------------------------------------------------------------------
    return ++this->cRef;
}

static ULONG OutsideRelease (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);
```

```
//-------------------------------------------------------------------
#ifdef DEBUG
    if (--this->cRefs[INTERFACE_INDEX (this, pfoo)] < 0)
        ASSERT (FALSE);
#endif
\\-------------------------------------------------------------------
    if (--this->cRef == 0)
    {
        --vcObjects;

        Free (this);
        return 0;
    }
    return this->cRef;
}
```

C++ techniques

There are two major techniques for implementing multiple interfaces in C++, which may be used separately or in combination: multiple inheritance and nested classes.

Multiply inherit each interface

The most direct method is to multiply inherit from all of the interfaces. Much of the time this is very convenient:

```
class COutside : IFoo, IBaz
{
    HRESULT __stdcall QueryInterface (REFIID riid, void **ppv);
    ULONG __stdcall AddRef (void) { return ++cRef; };
    ULONG __stdcall Release (void);

    // IFoo methods
    HRESULT __stdcall SetValue (int value);
    HRESULT __stdcall GetValue (int *pvalue);

    // IBaz methods
    HRESULT __stdcall SquareValue ();

    int cRef;
    int value;
};

HRESULT COutside::QueryInterface (REFIID riid, void **ppv)
{
    if (riid == IID_IFoo)
        *ppv = (IFoo *)this;
    else if (riid == IID_IBaz)
        *ppv = (IBaz *)this;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }
    AddRef ();
    return NOERROR;
}

ULONG COutside::AddRef (void);
{
    return ++cRef;
};

ULONG COutside::Release (void);
{
    if (--cRef == 0)
    {
        delete this;
        return 0;
    }
```

```
    return cRef;
};
```

At times, there will be clashes between members from different interfaces that have the same name and signature. If a different implementation is required for these, a different technique is required. By default, C++ will generate adjuster thunks to call one implementation from every vtable.

One alternative is to rename one of the interface methods by using a macro:

```
#define ClashingName FooClashingName
#include "ifoo.h"
#define ClashingName BazClashingName
#include "ibaz.h"
#undef ClashingName
```

Because the method names are never used outside the implementation, this causes no problems. COM defines a *binary* standard for vtables—the method names used by the server are not significant to clients.

There's another problem. When you are looking at the implementation of a method, you have no clue as to which interface it belongs to. Unless you've renamed it, you have to use its declared name. The class that it's attached to will be your outer implementation class, not the name of an interface. Thus it's advisable to use a comment convention to group methods for interfaces.

Nested classes

An alternative is to use a nested class for one or more of the interfaces. A situation where this commonly occurs is when different implementations are required for the **IUnknown** methods on different interfaces. This happens when an object is aggregatable, as described below in the section on aggregation. It is desirable for most classes to be aggregatable, so this may be expected to be common.

With this technique, each interface is implemented by a nested class. The outer class (**COutside** below) contains the instance data. The nested class code must be coupled to the outer class using either a back pointer or, preferably, the **IMPL** macro described earlier. A drawback of nested classes is that the C++ convenience of an implied **this** pointer at every reference is lost, causing the code to look very like the C equivalent. There's also no way to avoid the **IUnknown** delegation, as is possible in C. The Microsoft Foundation Class Library (MFC) uses nested classes with an equivalent to the **IMPL** macro for its OLE implementation.

The examples in *Inside OLE* use dynamically allocated interface implementations. This is somewhat like the nested class approach, but has more overhead. This dynamic allocation is fine for pedagogical purposes because it is clear and easy to understand, but I would not recommend it for shipping code.

Here's an example that shows the nested class approach:

```
struct COutsideCFoo : IFoo
{
    HRESULT    __stdcall QueryInterface (int iid, void **ppv);
    ULONG      __stdcall AddRef ();
    ULONG      __stdcall Release ();

    HRESULT    __stdcall SetValue (int);
    HRESULT    __stdcall GetValue (int *pvalue);
};

struct COutsideCBaz : IBaz
{
    HRESULT    __stdcall QueryInterface (int iid, void **ppv);
    ULONG      __stdcall AddRef ();
    ULONG      __stdcall Release ();

    HRESULT __stdcall SquareValue ();
};

struct COutside
{
    COutsideCFoo ifoo;
```

```
    COutsideCBaz ibaz;

    int cRef;
    int value;
};
......
HRESULT COutsideCBaz::SquareValue ()
{
    COutside *This = IMPL (COutside, ibaz, this);
    This->value *= This->value;
    return NOERROR;
}
```

Note that in C++, member functions have "extern" scope, so all member functions in the code must have unique names. This is so that the linker can populate vtables for derived classes in different compilation units (which is never needed with COM). For this reason, either the interface implementation classes must be nested within the **COutside** class as truly nested classes, or their names must be made globally unique, as shown here.

Implementing interface-specific reference counts in C++

With nested classes, interface-specific reference counts are straightforward. Because we will be implementing distinct **IUnknown** methods that typically delegate to a distinguished implementation, each implementation can place a reference count in its nested class.

With multiple inheritance, this does not work. An alternative here is to introduce an intermediate implementation for the **IUnknown** methods that calls some renamed **IUnknown** methods that the main class implements:

```
struct COutsideCIFoo : IFoo
{
    STDMETHOD (DebugQueryInterface) (REFIID riid, void **ppv) = 0;
    STDMETHOD_(ULONG, DebugAddRef) () = 0;
    STDMETHOD_(ULONG, DebugRelease) () = 0;

    STDMETHODIMP QueryInterface (REFIID riid, void **ppv)
    {
        return DebugQueryInterface (riid, ppv);
    };
    STDMETHODIMP_(ULONG) AddRef ()
    {
        ++cRef;
        return DebugAddRef ();
    }
    STDMETHODIMP_(ULONG) Release ()
    {
        if (--cRef < 0)
            ASSERT (FALSE);
        return DebugRelease ();
    }
    int cRef;
};

struct COutsideCIBaz : IBaz
{
//..... same as above
};

struct COutside : COutsideCIFoo, COutsideCIBaz
{
    STDMETHODIMP DebugQueryInterface (REFIID riid, void **ppv)
    {
        if (riid == IID_IFoo)
            *ppv = (IFoo *) this;
        else if (riid == IID_IBaz)
            *ppv = (IBaz *) this;
        else if (riid == IID_IUnknown)
            *ppv = (IPersist *) this;
        else
        {
            *ppv = 0;
```

```
            return E_NOINTERFACE;
        }
        DebugAddRef ();

        return NOERROR;
    };
    STDMETHODIMP_(ULONG) DebugAddRef ()
    {
        return ++cRef;
    }
    STDMETHODIMP_(ULONG) DebugRelease ()
    {
        return --cRef;
    }
    int cRef;

    // IFoo methods
    ...
    // IBaz methods
};
```

Much of the bulk here can be removed by using macros; because the **CIFoo** and **CIBaz** classes are nearly identical, they can be reduced to:

```
    DECLARE_DEBUG_IMPLEMENTATION (COutsideCIFoo, IFoo);
    DECLARE_DEBUG_IMPLEMENTATION (COutsideCIBaz, IBaz);
```

In nondebug code, this would be declared:

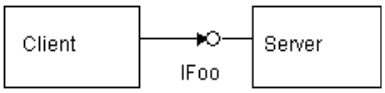
```
    #define DECLARE_DEBUG_IMPLEMENTATION(COutsideCIFoo, IFoo) \
        struct COutsideCIFoo : IFoo {}
```

Further, we would have a header file somewhere containing:

```
    #ifndef DEBUG
    #define DebugQueryInterface QueryInterface
    #define DebugAddRef AddRef
    #define DebugRelease DebugRelease
    #endif // DEBUG
```

Connection and Composition Paradigms

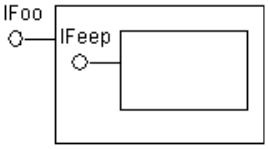
A very common paradigm with COM is for two objects to be connected. One object holds an interface to another. There are even diagrammatic methods of describing this:



The diagram indicates that the client holds an interface pointer to the **IFoo** interface on the server. When the client is given its **IFoo**, it calls **AddRef** on it, to hold the interface in existence. When the client no longer needs the **IFoo**, it releases it. The client should certainly do that when it is released.

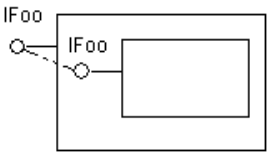
Containment

It is often useful when implementing a COM object to be able to use other COM objects as part of the implementation. This is straightforward. Such an object can be created during the outer object's creation function, and released during the outer object's **Release** function. Interfaces from the inner object can be used to assist in the implementation of interfaces of the outer object. This kind of containment reuse is common. Generally the contained object is not visible outside: if it is, then it would be best described as a connection relationship.



Delegation

When a new object is implemented that supports a known interface suite, a good reuse of code is to use an existing class to implement some aspects of an outer class. Conceptually, the simplest way to do this is to contain the existing class, implement all the required interfaces, and *delegate* calls on the interfaces to the interfaces of the existing object, where appropriate.



For example:

```
static HRESULT SomeMethod (ISomeInterface *psi, int a, int b, int c)
{
    SomeClass *this = IMPL (SomeClass, isi, psi);

    return this->pold->lpVtbl->SomeMethod (this->pold, a, b, c);
}
```

This is a powerful technique: The new class can retain the richness of the previous class. It can insert code before or after the call to the existing method, or simply not call the existing implementation for some methods. As far as the client is concerned, the new object is indistinguishable from the old one, so it happily continues providing its old behavior.

One thing to be aware of when doing this is that the set of interfaces that a client and object share represents a *service agreement* between the two objects. This kind of contract is sometimes referred to as a *type*. Clearly, COM objects can have multiple types. The OLE embedding interfaces, **IOleObject**, **IDataObject**, and **IPersistStorage** taken together form such a type. In this particular scenario, the requirements are fairly relaxed, and most violations would simply show up as unexpected user interface behavior. In other scenarios, breaking a service agreement may cause more serious problems.

If the existing object is later changed to support additional services, the containing object will continue to support only the old services. It's important to realize that if you modify the way that the contained object is called during the delegation, you must understand the services of which the interface is a part. If you interfere with the service agreement between two objects without understanding it fully, the two objects will likely not interoperate correctly.

Sometimes you may wish to modify the behavior of some interfaces, but leave others alone. When this happens, implementing trivial delegators for all those other interfaces becomes irksome. It will also start using significant amounts of code space. If you have several layers of nested reuse, the cost of the extra function call at each level will begin to mount up and your system will become large and slow. The traditional solution to this has been to start collapsing some of the layers of abstraction by opening up the old class and copying the code out of it. This wins locally, by avoiding function indirections and delegator implementations, but loses globally because you end up with multiple, slightly different copies of implementations, which tend to increase your working set (generally more than the delegators would), and which have new bugs. Trivial delegators don't tend to have bugs. Fortunately, COM supports some better methods.

The Universal Trivial Delegator

A Universal Trivial Delegator can be built that with a single implementation can delegate any interface with a much lower overhead than the C version above. Changing the compiler to support tail-call optimization would enable it to generate equivalent code, but would not achieve a single implementation for multiple interfaces. The example below uses some assembly code to share the stack frame and adjust the interface pointer, and does so with five instructions per method. Generally we want the **IUnknown** and other methods to be treated by different pieces of code, so the delegators separate the delegation into two targets. The **IUnknown** entries in the various "Other" interfaces below are never used by the delegators.

Clearly this is a generic piece of code that should be written once and shared by a number of groups. It is platform-specific and should be shipped with an SDK.

The C declarations look like this:

```
typedef struct
{
    struct _IDelegatorVtbl *lpVtbl;
} IDelegator;

typedef struct _IDelegatorVtbl
{
    int (*fn[64])(IDelegator *pdel);
} IDelegatorVtbl;
// ....
typedef struct
{
    IDelegator idel;
    IUnknown iunkUnknown;
    IUnknown iunkOther;
} Delegator;

typedef struct
{
    IDelegatorIndirect idel;
    IUnknown *punkUnknown;
    IUnknown *punkOther;
} IndirectDelegator;

typedef struct
{
    IDelegator idel;
    IUnknown iunkUnknown;
    IUnknown *punkOther;
} MethodIndirectDelegator;

typedef struct
{
    IDelegatorIndirect idel;
    IUnknown *punkUnknown;
    IUnknown iunkOther;
} UnknownIndirectDelegator;

extern IDelegatorVtbl VtblDelegator;
extern IDelegatorIndirectVtbl VtblIndirectDelegator;
extern IDelegatorMethodIndirectVtbl VtblMethodIndirectDelegator;
extern IDelegatorUnknownIndirectVtbl VtblUnknownIndirectDelegator;
```

The Intel® assembly code for each (direct) **IUnknown** method looks like this:

```
_del$ = 8
name2 PROC NEAR
    mov    eax, DWORD PTR _del$[esp-4]    ; pull interface pointer from stack
    add    eax, 4                         ; adjust by offsetof(Delegator, iunkUnknown)
    mov    DWORD PTR _del$[esp-4], eax    ; return interface pointer to stack
    mov    eax, DWORD PTR [eax]           ; load the lpVtbl pointer from it
    jmp      DWORD PTR [eax+offset*4]     ; jump indirect to to the member function at offset
name2 ENDP
```

The indirect code for each **IUnknown** method looks like this:

```
_del$ = 8
name PROC NEAR
    mov    eax, DWORD PTR _del$[esp-4]     ; pull interface pointer from stack
    mov    eax, DWORD PTR [eax+4]          ; pull delegating interface pointer
                                            (punkUnknown) from just after it
    mov    DWORD PTR _del$[esp-4], eax     ; return interface pointer to stack
    mov    eax, DWORD PTR [eax]            ; load the lpVtbl pointer from it
    jmp     DWORD PTR [eax+offset*4] ]     ; jump indirect to to the member
                                              function at offset
name ENDP
```

To use these delegators, include one in the class structure, initialize its **idel** member with the appropriate **VtblDelegator**, and

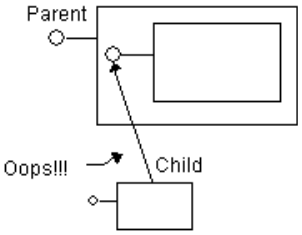
fill in either the **IUnknownVtbl** or **IUnknown** pointers with your own vtable or interface pointer as appropriate.

An example follows. Here we arrange to create an object of class **CLSID_Inside** to provide the implementation of the **IFeep** interface of **COutside**. Here, we want the **IUnknown** methods of **IFeep** to delegate to the **IUnknown** implementation of **COutside**, and the other methods to delegate to the **IFeep** interface of **CInside**. We arrange to **CoCreateInstance** a **CLSID_Inside** in the creation function, and release it in the **OutsideRelease** member. We set up the delegator structure in **COutside** and initialize it in the creation function. The delegator structure is an interface implementation, and we can hand out its address as one of our interfaces, because the **IUnknown** methods delegate to our own **IUnknown** implementation:

```
#include "ifoo.h"
#include "ibaz.h"
#include "ifeep.h"
#include "inside.h"
#include "util.h"
#include "idelegat.h"

typedef struct
{
    IFoo ifoo;
    IBaz ibaz;
    int cRef;
//------------------------------------------------------------------------
    IndirectDelegator del;
    IFeep *pfeep;
\\------------------------------------------------------------------------
    int value;
} COutside;

static HRESULT OutsideQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    if (IsEqualIID (riid, &IID_IUnknown) ||
            IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
    else if (IsEqualIID (riid, &IID_IBaz))
        *ppv = &this->ibaz;
//------------------------------------------------------------------------
    else if (IsEqualIID (riid, &IID_IFeep))
        *ppv = &this->del.idel;
\\------------------------------------------------------------------------
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG OutsideAddRef (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    return ++this->cRef
}

static ULONG OutsideRelease (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    if (--this->cRef == 0)
    {
        --vcObjects;
//------------------------------------------------------------------------
        Release (this->pfeep);
\\------------------------------------------------------------------------

        Free (this);
        return 0;
```

```
    }
    return this->cRef;
}
........

HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;

    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

    this->ifoo.lpVtbl = &vtblFoo;
    this->ibaz.lpVtbl = &vtblBaz;
    this->cRef = 1;

    CoCreateInstance (&CLSID_Inside, CLSCTX_SERVER, NULL, &IID_IFeep, &this-
                      >pfeep);

//---------------------------------------------------------------------
    this->del.idel = VtblIndirectDelegator;
    this->del.punkUnknown = (IUnknown *)&this->ifoo;
    this->del.punkOther = this->pfeep;
\\---------------------------------------------------------------------


    this->value = 0;
    ++vcObjects;
    hr = QueryInterface (&this->ifoo, riid, ppv);
    Release (&this->ifoo);
    return hr;
}
```

Note Not all interfaces *should* be trivially delegated this way. Some objects have interfaces that hand out interface pointers for other objects that they manage. Think of the object as a "parent" and the handed-out interface as being a "child." When the child is created, it may be given an interface to its parent, from which it can get some services. If the object wrapping the parent trivially delegates this interface, it will hand out a child who thinks its parent is the inner object. If some external code asks the child for its parent, it will then start invoking the old implementation of the parent's behavior. At the very least, it will not be able to get to the new behavior.



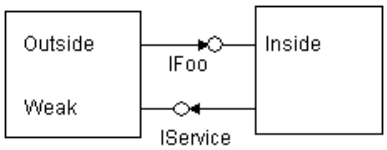
While this doesn't break any COM rules, it's unlikely to be what the interface specifications for the parent and child intended.

If this is a problem, the outer parent can create an outer child to contain the inner parent's child. This obviously requires a little more than trivial delegation.

Split Identities

Passing service interfaces to a contained object

At times, we need to deal with an inside object that needs some service from its outside object. Here the inside object cannot be passed an interface pointer to the outside object, as this would set up a cycle of references.



The best way to deal with this is to arrange for the outer object to have two object identities accessing the same data. Although the object has two identities, because they share an implementation they are not **AddRef**ing one another, so no reference cycle occurs. The identities have separate reference counts: the strong identity will cause the outside object to "shut down" when it is finally released. The way to think about shutdown is that the outside object makes some change that will break the reference cycle. Typically this involves signaling any objects holding the weak identity to let go of it. In our example, **Outside** will release the inside object, which will cause it to let go of the weak identity when it closes. Any other objects may let go of the weak identity some considerable time later: The shared data does not actually go away until both reference counts reach zero. For more information on "strong" versus "weak" references, see the "Managing Object Lifetimes in OLE" technical article in the MSDN Library.

The inner object needs to understand that it does not have a reference to the "real" outside/parent identity, and would need some additional service to obtain it transiently. There is not presently a standard interface for doing this, although some work is under way to establish one.

For example, suppose the inner object needs the **IService** interface, and the outer object wishes to provide it. Here's how that can be done. **IService** is another identity within **COutside**. Its **AddRef** and **Release** implementations manage a separate "weak" reference count. The outer object is freed only when both counts are zero, but can initiate that process when the "strong" count reaches zero, by releasing the inner object.

Some care is required with **Release** to deal with possible reentrancy: When **FooRelease** calls **Release** (**this->pfeep**), below, the inner object will turn around and release its **IService** interface. This will call **SvcRelease**, which would free the outside object if the weak reference count were zero. To prevent this, we bracket the release of weak reference holders with an extra **AddRef/Release** pair on the weak identity, which controls the resource freeing. This delays the deallocation until the outside object is ready:

```
#include "util.h"
#include "ifoo.h"
#include "ibaz.h"
//------------------------------------------------------------------------
#include "ifeep.h"
#include "iservice.h"
#include "inside.h"
\\------------------------------------------------------------------------

typedef struct
{
    IFoo ifoo;
    IBaz ibaz;
    int cRef;
    int value;
    IFeep *pfeep;

//------------------------------------------------------------------------
    IService iservice;
    int cRefWeak;
\\------------------------------------------------------------------------
} COutside;
.....
//------------------------------------------------------------------------
static ULONG FooRelease (IBaz *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo)
    if (--this->cRef == 0)
    {
        SvcAddRef (&this->iservice);
        if (this->pfeep)
                Release (this->pfeep);
        SvcRelease (&this->iservice);
        return 0;
    }
    return this->cRef;
```

```
}
\\---------------------------------------------------------------------

......
//---------------------------------------------------------------------
static HRESULT SvcQueryInterface (IService *psvc, REFIID riid, void **ppv)
{
    COutside *this = IMPL (COutside, iservice, psvc);

    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IBaz))
        *ppv = &this->iservice;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG SvcAddRef (IService *psvc)
{
    COutside *this = IMPL (COutside, iservice, psvc);
    return ++this->cRefWeak;
}

static ULONG SvcRelease (IService *psvc)
{
    COutside *this = IMPL (COutside, iservice, psvc);
    if (--this->cRefWeak == 0 && this->cRef == 0)
    {
        Free (this);
        return 0;
    }
    return this->cRefWeak;
}

static HRESULT SvcExample (IService *psvc)
{
    COutside *this = IMPL (COutside, iservice, pbaz);
    // ???
    return NOERROR;
}

static IServiceVtbl vtblService =
{
    (QITYPE) SvcQueryInterface,
    (ARTYPE) SvcAddRef,
    (RLTYPE) SvcRelease,
    SvcExample
};
\\---------------------------------------------------------------------
......
HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;

    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

    this->ifoo.lpVtbl = &vtblFoo;
    this->ibaz.lpVtbl = &vtblBaz;
    this->cRef = 1;
//---------------------------------------------------------------------
    this->iservice.lpVtbl = &vtblService;
    this->cRefWeak = 0;
```

```
\\--------------------------------------------------------------------

    this->value = 0;
    hr = CoCreateInstance (&CLSID_Inside, NULL, CLSCTX_INPROC_SERVER,
                           &IID_IFeep, &this->pfeep);

//--------------------------------------------------------------------
    if (hr == NOERROR)
    {
        this->pfeep->lpVtbl->SetService (this->pfeep, &this->iservice);
\\--------------------------------------------------------------------

        hr = QueryInterface (&this->ifoo, riid, ppv);
    }
    ++vcObjects;

    Release (&this->ifoo);

    return hr;
}
```

Mutual references between multiple contained objects are not a problem because the container is in control of their lifetime and can break the cycle when it decides to shut down. References between contained objects do not hold their container alive.

Another scenario in which split identities work well is in notification. The outside object may be changing the state of some object it contains, but may wish to be notified of the effect of such changes. This is particularly important if other clients can change the object's state (as would happen in a connection scenario), or if the outcome of the state change would otherwise be difficult to predict.

Implementing split identities with many common interfaces

Any interfaces exposed by outside that are passed to inside must be delegated to avoid a reference cycle. If the interfaces are large or called often, this is not an attractive solution. The Universal Trivial Delegator can be used again to avoid adding delegator code, but if the inner object requires most or all of the interfaces that the outer object provides, we will have to provide two implementations of all our interfaces.

Fortunately, there is another way to implement split identities that avoids delegation. Let's assume that the inner object may wish to **QueryInterface** for other interfaces that the outer supports (excluding **IFeep** . . .).

This technique accesses the state of **COutside** through an indirection, **pOutsideThis**. It costs an extra indirection that most existing COM implementations are already paying by using back pointers. We can still share the **IUnknown** implementation by making it slightly smarter. We rely on **pOutsideThis** being the first element of the **COutsideIdentity** structure so that the **SPLIT_IMPL** macro can find it without depending on its name. This method is much cheaper than even the Universal Trivial Delegator discussed earlier. Here is the implementation of the **SPLIT_IMPL** macro:

```
#define SPLIT_IMPL(outerClass, innerClass, member, pointer) \
    (*(outerClass **) IMPL (innerClass, member, pointer))
```

It works by first using the **IMPL** macro to find the top of the structure containing the interfaces and then assuming that the first member of that structure is a back pointer to the **Outside** class structure. **FIND_SPLIT_IMPL**, the split identity equivalent of **FindImpl**, is similar. It uses **FindImpl** to find the first vtable and then finds the top of structure using offsets:

```
#define FIND_SPLIT_IMPL(innerClass, iFirst, vtblFirst, pointer) \
    ((innerClass *) (((char  *)FindImpl (pointer, vtblFirst)) - offsetof (innerClass, iFirst)))
```

Notice that in the example below, **IBaz** is supported only on the strong identity because it requires the inner object, which will be removed when there are no longer any strong references. It is up to the split object to ensure that the weak identity has stable behavior after the strong identity has been released. There is a vtable pointer for each split interface on both identities, even if one of them isn't used. If this is too much cost, then they can be placed in **COutside** with more coding complexity.

Here is FOOSPLIT.C:

```
#include "util.h"
#include "ifoo.h"
#include "ibaz.h"
```

```
#include "ifeep.h"
#include "iservice.h"
#include "inside.h"

typedef struct COutside
{
//--------------------------------------------------------------------
    struct _identity {
        struct COutside *pOutsideThis;
\\--------------------------------------------------------------------
        IFoo ifoo;
        IBaz ibaz;
        IService iservice;
        int cRef;
//--------------------------------------------------------------------
    } strong, weak;
\\--------------------------------------------------------------------
    int value;
    IFeep *pfeep;
} COutside;

//--------------------------------------------------------------------
typedef struct _identity COutsideIdentity;
\\--------------------------------------------------------------------
static IFooVtbl vtblFoo;

static HRESULT OutsideQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
//--------------------------------------------------------------------
    COutsideIdentity *thisId = FIND_SPLIT_IMPL(COutsideIdentity, ifoo, &vtblFoo,
                                               pfoo);
    COutside *this = thisId->pOutsideThis;

\\--------------------------------------------------------------------
    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IFoo))
        *ppv = &thisId->ifoo;
    else if (IsEqualIID (riid, &IID_IBaz) && thisId == &this->strong)
        *ppv = &thisId->ibaz;
    else if (IsEqualIID (riid, &IID_IService) && thisId == &this->weak)
        *ppv = &thisId->iservice;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG OutsideAddRef (IFoo *pfoo)
{
//--------------------------------------------------------------------
    COutsideIdentity *thisId = FIND_SPLIT_IMPL(COutsideIdentity, ifoo, &vtblFoo,
                                               pfoo);
\\--------------------------------------------------------------------
    return ++thisId->cRef;
}

static ULONG OutsideRelease (IFoo *pfoo)
{
//--------------------------------------------------------------------
    COutsideIdentity *thisId = FIND_SPLIT_IMPL(COutsideIdentity, ifoo, &vtblFoo,
                                               pfoo);
    COutside *this = thisId->pOutsideThis;

    if (--thisId->cRef != 0)
        return thisId->cRef;
    if (thisId == &this->strong)
    {
        AddRef (&this->weak.ifoo);
        if (this->pfeep)
```

```
                Release (this->pfeep);
            Release (&this->weak.ifoo);
        }
        else // weak identity
        {
            if (this->strong.cRef == 0)
            {
                    --vcObjects;

                    Free (this);
            }
        }
        return 0;
}
\\--------------------------------------------------------------------

static HRESULT SetValue (IFoo *pfoo, int value)
{
//--------------------------------------------------------------------
    COutside *this = SPLIT_IMPL (COutside, COutsideIdentity, ifoo, pfoo);
\\--------------------------------------------------------------------
    this->value = value;
    return NOERROR;
}

static HRESULT GetValue (IFoo *pfoo, int *pValue)
{
//--------------------------------------------------------------------
    COutside *this = SPLIT_IMPL (COutside, COutsideIdentity, ifoo, pfoo);
\\--------------------------------------------------------------------
    if (!pValue)
        return E_POINTER;
    *pValue = this->value;
    return NOERROR;
}

static IFooVtbl vtblFoo =
{
    (QITYPE) OutsideQueryInterface, (ARTYPE) OutsideAddRef, (RLTYPE)

                                                        OutsideRelease,
    SetValue,
    GetValue
};

static HRESULT SquareValue (IBaz *pbaz)
{
//--------------------------------------------------------------------
    COutside *this = SPLIT_IMPL (COutside, COutsideIdentity, ibaz, pbaz);
\\--------------------------------------------------------------------
    this->value *= this->value;
    this->pfeep->lpVtbl->Sum (this->pfeep, this->value);

    return NOERROR;
}

static IBazVtbl vtblBaz =
{
    (QITYPE) OutsideQueryInterface, (ARTYPE) OutsideAddRef, (RLTYPE)

                                                       OutsideRelease,
    SquareValue
};

static HRESULT SvcExample (IService *psvc)
{
//--------------------------------------------------------------------
    COutside *this = SPLIT_IMPL (COutside, COutsideIdentity, iservice, psvc);
\\--------------------------------------------------------------------
    // ???
    return NOERROR;
}
```

```
static IServiceVtbl vtblService =
{
    (QITYPE) OutsideQueryInterface, (ARTYPE) OutsideAddRef, (RLTYPE)

                                                        OutsideRelease,
    SvcExample
};


HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    if (punkOuter)
         return CLASS_E_NOAGGREGATION;
    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

//------------------------------------------------------------------
    this->strong.pOutsideThis = this;
    this->strong.ifoo.lpVtbl = &vtblFoo;
    this->strong.ibaz.lpVtbl = &vtblBaz;
    this->strong.iservice.lpVtbl = &vtblService;
    this->strong.cRef = 1;
    this->weak = this->strong;
    this->weak.cRef = 0;
\\------------------------------------------------------------------

    this->value = 0;
//------------------------------------------------------------------
    hr = CoCreateInstance (&CLSID_Inside, NULL, CLSCTX_INPROC_SERVER,
                           &IID_IFeep, &this->pfeep);
\\------------------------------------------------------------------
    if (hr == NOERROR)
    {
        this->pfeep->lpVtbl->SetService (this->pfeep, &this->weak.iservice);
        hr = QueryInterface (&this->strong.ifoo, riid, ppv);
    }
    ++vcObjects;

    Release (&this->strong.ifoo);

    return hr;
}
```

The same example in C++

Up until now, the C++ samples have looked clearer, mostly by virtue of hiding the **this** pointer. For this example, that is no longer feasible, as we have to introduce a back pointer to access the shared state. The C++ methods look very like the equivalent C code. I've highlighted the actual member functions, since in a real example they will constitute the bulk of the code. Here is FOOSPLIT.CPP:

```
#include "ifoo.h"
#include "ibaz.h"
#include "ifeep.h"
#include "iservice.h"
#include "inside.h"

extern "C" int vcObjects;

class COutside;

struct COutsideIdentity : IFoo, IBaz, IService
{
// IUnknown methods
    ULONG __stdcall AddRef () { return ++m_cRef; };
// IFoo methods
    HRESULT __stdcall SetValue (int v);
    HRESULT __stdcall GetValue (int *pv);
```

```
// IBaz methods
    HRESULT __stdcall SquareValue ();

// IService methods
    HRESULT __stdcall Example ();

    COutside * m_pThis;
    int m_cRef;
};

struct COutsideStrong : COutsideIdentity
{
    HRESULT __stdcall QueryInterface (REFIID riid, void **ppv);
    ULONG __stdcall Release ();
};

struct COutsideWeak : COutsideIdentity
{
    HRESULT __stdcall QueryInterface (REFIID riid, void **ppv);
    ULONG __stdcall Release ();
};

struct COutside
{
    COutsideStrong m_strong;
    COutsideWeak m_weak;

    COutside ();
    ~COutside ();
    int m_value;
    IFeep * m_pfeep;
};

COutside::COutside ()
{
    m_weak.m_cRef = 0;
    m_weak. m_pThis = this;
    m_strong. m_cRef = 1;
    m_strong. m_pThis = this;
    ++vcObjects;
}

COutside:: ~COutside ()
{
    --vcObjects;
}

// Strong IUnknown methods

HRESULT COutsideStrong::QueryInterface (REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown || riid == IID_IFoo)
        *ppv = (IFoo *) this;
    else if (riid == IID_IBaz)
        *ppv = (IBaz *) this;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    ((IUnknown *)*ppv)->AddRef ();
    return NOERROR;
}

ULONG COutsideStrong::Release ()
{
    if (--m_pThis->m_strong.m_cRef == 0)
    {
        m_pThis->m_weak.AddRef ();
        m_pThis->m_pfeep->Release ();
        m_pThis->m_weak.Release ();
```

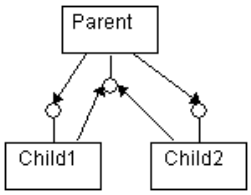
```
        return 0;
    }
    return m_pThis->m_strong.cRef;
}

// Weak IUnknown methods

HRESULT COutsideWeak::QueryInterface (REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown || riid == IID_IFoo)
        *ppv = (IFoo *) this;
    else if (riid == IID_IService)
        *ppv = (IService *) this;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    ((IUnknown *)*ppv)->AddRef ();
    return NOERROR;
}

ULONG COutsideWeak::Release ()
{
    if (--m_pThis-> m_weak.m_cRef == 0 && m_pThis->m_strong. m_cRef == 0)
    {
            --vcObjects;
            delete this;
            return 0;
    }
    return m_pThis-> m_weak.m_cRef;
}

//------------------------------------------------------------------
// IFoo methods

HRESULT COutsideIdentity:: SetValue (int v)
{
    m_pThis-> m_value = v;
    return NOERROR;
}

HRESULT COutsideIdentity::GetValue (int *pv)
{
    if (!pv)
        return E_POINTER;
    *pv = m_pThis-> m_value;
    return NOERROR;
}

// IBaz methods

HRESULT COutsideIdentity::SquareValue ()
{
    m_pThis-> m_value *= m_pThis-> m_value;
    m_pThis-> m_pfeep->Sum (m_pThis-> m_value);
    return NOERROR;
}

// IService methods

HRESULT COutsideIdentity::Example ()
{
    // ???
    return NOERROR;
}

\\------------------------------------------------------------------
extern "C" HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    HRESULT hr;
```

```
    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;
    COutside *pout = new COutside;
    hr = CoCreateInstance (CLSID_Inside, NULL, CLSCTX_INPROC_SERVER, IID_IFeep,
                                                    (void **)&pout->m_pfeep);
    if (hr == NOERROR)
    {
        pout->m_pfeep->SetService ((IService *)&pout->m_weak);
        hr = pout->m_strong.QueryInterface (riid, ppv);
    }

    pout-> m_strong.Release ();
    return hr;
}
```

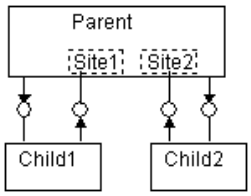
There is in fact a way to hide the back pointer again in C++, by inheriting the strong and weak identities and moving the instance data into a virtual base class. Unfortunately, the code generated for virtual base classes is more general than necessary for this context, and the required multiple inheritance nesting breaks the compiler. I dare say it would break the more casual readers also.

Reference cycles and part-whole hierarchies

Another scenario in which the reference cycle issue commonly comes up is in part-whole hierarchies. Here we have a tree, where each node is a COM object. To facilitate navigation around the tree and to obtain contextual information, nodes in the tree will usually require references to their parents as well as to their children. This thus indicates a reference cycle between parent and child. There are two ways to deal with this problem: Either you avoid keeping references to the parent node, and instead have some other means to obtain the parent, or you keep references, but inform the parent when you **AddRef** or **Release** the item so that it knows that the reference is from a child, and thus should not prevent shutdown.



In many situations the child's view of its parent is different from one of the parent's peers, so a split identity is appropriate. If the child needs or is asked for the parent's identity, it can use a *service* from the parent to obtain that identity.

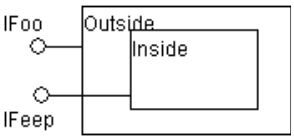


Symmetric lifetime management

In some situations it is necessary to have two mutually dependent objects that have different identities, but share a common lifetime. The requirement is that a reference to either identity will hold both objects alive and that when those references are released, that both objects are freed. This necessarily involves sharing a reference count. Solutions to this problem will be discussed in a later document (also see the "Managing Object Lifetimes in OLE" technical article).

Aggregation

In the section on delegation, we discussed ways of arranging for an interface on a contained object to appear on its container. While the Universal Trivial Delegator is much smaller and faster than explicit delegator functions, there is another technique available to the COM programmer.



Rather than delegating an entire interface trivially, it would be very tempting simply to hand out the inner object's interface pointers in response to the outer object's **QueryInterface**. Unfortunately, this breaks the COM rules. The inner object's **IUnknown** methods do not know about the outer object, and the inner **QueryInterface** would not be able to succeed or fail the same way as the outer **QueryInterface** on the exposed interface, and would not be able to return the outer object's identity. This may not be a problem for a particular client of the outer COM object, but it will be for some—particularly the remoting code. Such an object may be usable in a limited way, but it is not a COM object.

Aggregating another object

There is a way to make this work, however: The inner object can be given the **IUnknown** pointer of the outer object in its **IClassFactory::CreateInstance** call. It then delegates its **IUnknown** methods to that pointer, much as other interfaces do in a single object. This is called *COM aggregation*.

Here we assume the existence of a class **Inside**, which supports **IUnknown** and **IFeep**, which we want to expose as an interface on **COutside**. **COutside** also requires the **IFeep** interface to implement its **IBaz** interface. This is just an example; not all aggregation outer objects need internal services from their inner objects. **COutside** requests the **IFeep** interface from the inner object and places it in its class structure. This is called *caching* an interface pointer. When it does so, it releases *itself* to balance the extra reference count. During shutdown, **Outside** will **AddRef** itself before releasing **this->pfeep** to balance the reference counts the other way. It performs and stabilizes the reference count to 1 before this to prevent the **OutsideRelease** method being reentered when **this->pfeep** is released:

```
#include "util.h"
#include "ifoo.h"
#include "ibaz.h"
//------------------------------------------------------------------------
#include "ifeep.h"
#include "inside.h"
\\------------------------------------------------------------------------

typedef struct
{
    IFoo ifoo;
    IBaz ibaz;
    int cRef;
    int value;
//------------------------------------------------------------------------
    IUnknown *punkInside;
    IFeep *pfeep;
\\------------------------------------------------------------------------
} COutside;

static IFooVtbl vtblFoo;

static HRESULT OutsideQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    if (IsEqualIID (riid, &IID_IUnknown) ||
            IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
    else if (IsEqualIID (riid, &IID_IBaz))
        *ppv = &this->ibaz;
//------------------------------------------------------------------------
    else if (IsEqualIID (riid, &IID_IFeep))
        *ppv = this->pfeep;
\\------------------------------------------------------------------------
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }
```

```
    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG OutsideAddRef (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);
    return ++this->cRef;
}

static ULONG OutsideRelease (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);
    if (--this->cRef == 0)
    {
//--------------------------------------------------------------------
        this->cRef = 1; // guard
        if (this->pfeep)
        {
                AddRef (&this->ifoo); // release cached pointer
                Release (this->pfeep); //        "
        }
        Release (this->punkInside);
\\--------------------------------------------------------------------

        Free (this);
        --vcObjects;
        return 0;
    }
    return this->cRef;
}

static HRESULT SetValue (IFoo *pfoo, int value)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);
    this->value = value;
    return NOERROR;
}

static HRESULT GetValue (IFoo *pfoo, int *pValue)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);
    if (!pValue)
        return E_POINTER;
    *pValue = this->value;
    return NOERROR;
}

static IFooVtbl vtblFoo =
{
    (QITYPE) OutsideQueryInterface,     (ARTYPE) OutsideAddRef,
    (RLTYPE) OutsideRelease,
    SetValue,
    GetValue
};

static HRESULT SquareValue (IBaz *pbaz)
{
    COutside *this = IMPL (COutside, ibaz, pbaz);
    this->value *= this->value;
//--------------------------------------------------------------------
    this->pfeep->lpVtbl->Sum (this->pfeep, this->value);
\\--------------------------------------------------------------------
    return NOERROR;
}

static IBazVtbl vtblBaz =
{
    (QITYPE) OutsideQueryInterface,     (ARTYPE) OutsideAddRef,
    (RLTYPE) OutsideRelease,
    SquareValue
};
```

```
HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;
    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

    this->ifoo.lpVtbl = &vtblFoo;
    this->ibaz.lpVtbl = &vtblBaz;
    this->cRef = 1;
    this->value = 0;
    this->pfeep = 0;

    ++vcObjects;
//--------------------------------------------------------------------
    hr = CoCreateInstance (&CLSID_Inside, (IUnknown *)&this->ifoo,
                      CLSCTX_INPROC_SERVER, &IID_IUnknown, &this->punkInside);
    if (hr != NOERROR)
    {
\\--------------------------------------------------------------------

        Release (&this->ifoo);
        return hr;
    }
//--------------------------------------------------------------------
    hr = QueryInterface (this->punkInside, &IID_IFeep, &this->pfeep);
    if (hr == NOERROR)
    {
            Release (&this->ifoo);
\\--------------------------------------------------------------------
            hr = QueryInterface (&this->ifoo, riid, ppv);
    }

    Release (&this->ifoo);
    return hr;
}
```

Here, then, are the rules for being an aggregator:

- The aggregator should pass its own **punkOuter** implementation to the object to aggregate as the *punkOuter* parameter of **CoCreateInstance** (or in other creation scenarios in which a *punkOuter* is required). This *punkOuter* will be the one it received as its *punkOuter* when it was created, or its own **IUnknown** implementation if it is not part of an aggregate.
- When creating the inner object, the aggregator must ask initially for the **IUnknown** interface. If it does not, it will have no way to control the inner object's lifetime.
- When caching a pointer to an interface on the inner object, the outer object should not release any interface it obtains for private use because the inner object may be keeping per-interface reference counts, and may invalidate the interface if it is released. This is a change to the OLE version 2.0 aggregation rules, which was done for 32-bit OLE specifically to allow this. Instead, to hold an interface pointer to an inner object, the outer object must release the **punkOuter** that was passed to the inner object. This is because the interface, though obtained from the inner object, now belongs to the outer object (**COutside**) and thus **AddRef**s that **punkOuter**. This prevents a reference cycle, though it does restrict an implementation of **QueryInterface** to adding one reference.
- When it closes down, the outer object should **AddRef** the **punkOuter** passed to its inner object, and release any held interface pointer from the inner object, thus enabling the per-interface reference counts in the inner object to be balanced.
- The aggregator must guard its shutdown by increasing its reference count again before releasing the aggregated object. This is to prevent the inner object causing the **Release** to be reentered when the inner releases any cached pointers.

A present limitation with aggregation is that the outer and inner objects have to be in the same process (and multithreading apartment for that matter), though it's possible that this restriction may be lifted in the future.

Being part of an aggregate

The rules of behavior for an object when it is an inner part of an aggregate are as follows:

- When the inner object receives a **QueryInterface** on one of its interfaces *other than **IUnknown***, it delegates it to its **punkOuter** (the second argument to **CoCreateInstance**). This way, when one of those interfaces is exposed outside the outer object, the same delegation occurs that the other interfaces on the outside object are already doing. The **QueryInterface** should always **AddRef** the pointer it is returning, not some other one.
- The inner object does not delegate the **QueryInterface** of its **IUnknown**, but instead returns one of its own interfaces, and **AddRef**s its **punkOuter** instead of its own reference count. This is because the interfaces of an aggregated object "belong" to the outer object.
- The inner object does not **AddRef** its **punkOuter** when it first passed it in, even though it is being stored by the object. This is to avoid a reference cycle between the outer and inner objects. Such a reference cycle would prevent either from being freed when all external references have gone away.
- If the aggregator does not initially ask for the **IUnknown** interface of the inner object, then the creation should fail.

So what does an object have to do differently to be aggregatable? Actually, very little. Although aggregatability is optional in COM, it costs so little that it makes sense to make most COM classes aggregatable unless there is a specific reason not to. After all, if you want to aggregate things, it makes sense to build things that are aggregatable.

There are a couple of differences from the earlier examples: It is no longer possible to share the **IUnknown** vtable with the first interface. This is because an aggregatable object behaves differently with **IUnknown** than with other interfaces. The **IUnknown** implementation must also be prepared to delegate to the **punkOuter** if it detects that it is not being called from the **IUnknown** interface. It does this detection by comparing the vtable pointers in **FindImpl**.

If you are using **IUnknown** delegators in the other interfaces, they can delegate directly to **punkOuter**. The **IUnknown** implementation would then not need to do this detection.

Being aggregatable thus costs 8 bytes per instance and 9 lines of code per class. For all but the smallest objects, this is worthwhile.

Here is INSIDE.C. This sample is pretty close to boilerplate for most of my own COM object implementations:

```
#include "util.h"
#include "ifeep.h"

typedef struct
{
//------------------------------------------------------------------------
    IUnknown iunk;
\\------------------------------------------------------------------------
    IFeep ifeep;
    int cRef;
//------------------------------------------------------------------------
    IUnknown *punkOuter;
\\------------------------------------------------------------------------
    int value;
    IService *psvc;
} CInside;

static IUnknownVtbl vtblUnknown;

static HRESULT UnkQueryInterface (IUnknown *punk, REFIID riid, void **ppv)
{
    CInside *this = FindImpl (punk, &vtblUnknown);

//------------------------------------------------------------------------
    if (punk->lpVtbl != &vtblUnknown && this->punkOuter)
        return QueryInterface (this->punkOuter, riid, ppv);
\\------------------------------------------------------------------------
    else if (IsEqualIID (riid, &IID_IUnknown))
        *ppv = &this->iunk;
    else if (IsEqualIID (riid, &IID_IFeep))
        *ppv = &this->ifeep;
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
```

```
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG UnkAddRef (IUnknown *punk)
{
    CInside *this = FindImpl (punk, &vtblUnknown);

//--------------------------------------------------------------------
    if (punk->lpVtbl != &vtblUnknown && this->punkOuter)
        return AddRef (this->punkOuter);
\\--------------------------------------------------------------------
    return ++this->cRef;
}

static ULONG UnkRelease (IUnknown *punk)
{
    CInside *this = FindImpl (punk, &vtblUnknown);

//--------------------------------------------------------------------
    if (punk->lpVtbl != &vtblUnknown && this->punkOuter)
        return Release (this->punkOuter);
\\--------------------------------------------------------------------
    if (--this->cRef == 0)
    {
        if (this->psvc)
                Release (this->psvc);

        Free (this);
        --vcObjects;
        return 0;
    }
    return this->cRef;
}

//--------------------------------------------------------------------
static IUnknownVtbl vtblUnknown =
{
    UnkQueryInterface, UnkAddRef, UnkRelease
};
\\--------------------------------------------------------------------



static HRESULT SetService (IFeep *pfeep, IService *psvc)
{
    CInside *this = IMPL (CInside, ifeep, pfeep);
    if (this->psvc)
        Release (this->psvc);
    this->psvc = psvc;
    if (this->psvc)
        AddRef (this->psvc);
    return NOERROR;
}

static HRESULT Sum (IFeep *pfeep, int value)
{
    CInside *this = IMPL (CInside, ifeep, pfeep);
    this->value += value;
    if (this->psvc)
        this->psvc->lpVtbl->Example (this->psvc);
    return NOERROR;
}

static HRESULT GetSum (IFeep *pfeep, int *pvalue)
{
    CInside *this = IMPL (CInside, ifeep, pfeep);

    if (!pvalue)
        return E_POINTER;
```

```
    *pvalue = this->value;

    return NOERROR;
}

static IFeepVtbl vtblFeep =
{
    (QITYPE)UnkQueryInterface, (ARTYPE)UnkAddRef, (RLTYPE)UnkRelease,
    SetService,
    Sum,
    GetSum
};

HRESULT CreateInside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    CInside *this;
    HRESULT hr;
    *ppv = 0;
    if (hr = Alloc (sizeof (CInside), &this))
        return hr;

    this->iunk.lpVtbl = &vtblUnknown;
    this->ifeep.lpVtbl = &vtblFeep;
    this->cRef = 1;
//----------------------------------------------------------------------
    this->punkOuter = punkOuter; // will be NULL when not aggregated
\\----------------------------------------------------------------------
    this->value = 0;
    this->psvc = 0;
    ++vcObjects;

    hr = QueryInterface (&this->iunk, riid, ppv);

    Release (&this->iunk);

    return hr;
}
```

More aggregation rules

There are some more usage rules about aggregation behavior:

- An inner object should not assume that the address of its **punkOuter** is the address of the identity **IUnknown** of the outer object. If it wants the identity of the outer object, it should **QueryInterface** for it (though see the following rule).
- An inner object should not obtain services for its own use from its outer object by calling **QueryInterface** on its **punkOuter**. There are a couple of reasons for this: one is that the outer object may actually not *want* the inner object to use that particular implementation, and will not be able to distinguish such a **QueryInterface** from one from outside itself. The other reason is that when aggregation is nested, the **QueryInterface** will delegate through to the outermost object, which might choose not to expose some interface on the middle object. If it does that, then the middle and innermost objects will be broken. Aggregation is not allowed to be visible from outside the outer object, and this behavior breaks that encapsulation rule.
- For similar reasons, an outer object should not pass its own interfaces (including its **punkOuter**) inwards to inner objects. If it does, the inner object will be obliged to **AddRef** them, which will cause a reference cycle. Because it can't tell where the interface came from, the inner object may also choose to keep other references to the interface, so the outer object will not know how many releases it would need to perform to avoid the cycle. A way to deal correctly with this situation is described below.

"Blind" delegation of QueryInterface

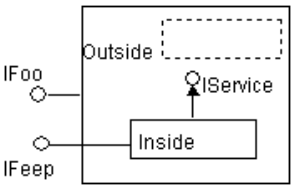
One possibility with aggregation is to delegate to an inner object the **QueryInterface** calls for all interfaces that are not understood by the outer object. This is tempting because it is easy, and gives an effect that is quite similar to inheritance in C++. The problem, as with C++, is that if the inner object (base class) changes, it may expose services that involve interfaces that the outer object already implements, and is thus obscuring. What then happens is that the outer object is interfering with services that it is not aware of. This can cause the outer object to appear to support a service that it does not then implement correctly. This is a *bug*. In C++ there is no way that a client can prevent this kind of problem when using inheritance—it is the default behavior.

It is unwise to delegate interfaces that you do not understand unless you also do not interfere with any other interfaces (meaning that you pass all other interfaces through unchanged, with only additional "before" and "after" code in the delegators). The only other case where this is safe is where you have knowledge or control over the future of the object you are aggregating. If the inner object is provided by someone else, then you will not generally have such assurances.

A place where this issue arises is in creating a custom handler using the OLE default handler. This is aggregated into custom handlers provided by some OLE embedding object implementors. If your custom handler could be broken by the default handler being exposed directly, or you are implementing interfaces that the default handler may later implement, you will need to avoid blind delegation.

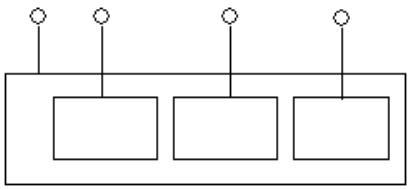
Passing service interfaces to an aggregated object

How then can an aggregated object obtain services from its outer object? The short answer is that an object should not use the fact that it is aggregated to obtain services that it would not otherwise have. It is up to the outer object to provide such services using an object with an identity different than its own. This is another case where split identities are useful. The inner object is aggregated with the strong identity of the outside object, but gets its service connection from the weak identity.



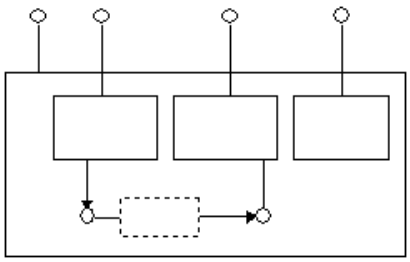
Multiple aggregation

Multiple aggregation is the COM equivalent of multiple inheritance (MI). Most reasonable people will avoid the multiple inheritance of implementations, mostly because MI doesn't scale linearly either in complexity or code space costs, and has dire consequences in the face of independently versioned classes. A C++ class that inherits multiply exposes all the classes it inherits from *and everything that every one of those classes inherits from, even indirectly*. Fortunately, multiple aggregation is somewhat more benign. With **QueryInterface**, the aggregator has complete control over the set of interfaces it exposes, so selecting from multiple aggregated objects is reasonable and safe.



Passing service interfaces between aggregated objects

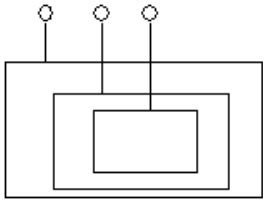
It is important to bear in mind that the aggregated objects each share the identity (and reference count) of their outer object, so passing an interface between them will immediately generate a reference cycle. A reasonable solution to this is again to use a split identity in the outer object, so that inner objects see interfaces on an identity other than the outer one. If inner objects need to use interfaces from one another, the simplest solution is to delegate those interfaces on the weak identity. The delegator uses a cached interface pointer to one of the other inner objects.



Nested aggregation

Nested aggregation occurs when an object aggregates something that is already an aggregate. Because aggregation is private,

the outermost object is fully isolated from the inner aggregation. The object in the middle has one extra concern: The **punkOuter** that it passes to the innermost object should be the **punkOuter** it receives from its own outer object, or should delegate to it. Nested aggregation is important because it provides a way to flatten multiple layers of abstraction without losing performance, and yet it does so in a way that is completely transparent to implementors.



Optimizations

As implementations become richer, they tend to support a large number of interfaces. As objects scale up they tend to require more resources and more initialization time. If the use of an object is simple and lightweight, it makes sense for the object to delay or avoid some initialization in order to provide better services for humbler customers. There are a number of techniques possible within the COM standard that enable this. It's important to note that these are just implementation techniques. The fact that they are being used is invisible outside the object involved.

Delayed Initialization

Many kinds of initialization can occur during **QueryInterface** for particular interfaces. Not much needs to be said about this, other than that because you know that none of the member functions of an interface can be called before a **QueryInterface** for that interface, there is an opportunity to do some delayed initialization for that interface or a collection of interfaces in the **QueryInterface** implementation. Members of those interfaces can then assume the initialization without having to test for it.

Delayed Aggregation

Large and complex objects often aggregate other objects to provide particular services. If those services are not always required, then creating the aggregated objects is not necessary.

In the example below the **punkInside** member is initialized to zero during **CreateFoo**, and released if nonzero during **FooRelease**. The aggregation of inside occurs when either **IFeep** or **IBaz** is requested by calling **InitInside**, because **IBaz** also uses the inner object. The advantage of this technique is that if **IFoo** is the only interface of interest to the client, the client doesn't pay the cost of loading the inside object. For large objects, this may be quite significant. Further, some initialization can be delayed until the service is actually required, which smoothes the responsiveness of the collection of COM objects.

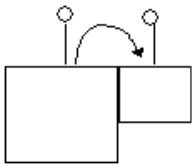
Here is the code:

```
typedef struct
{
    IFoo ifoo;
    IBaz ibaz;
    int cRef;
    int value;
    IUnknown *punkInside;
    IFeep *pfeep;
} COutside;

//------------------------------------------------------------------------
static HRESULT InitInside (COutside *this);
{
    HRESULT hr = CoCreateInstance (&CLSID_Inside, (IUnknown *)&this->ifoo,
       CLSCTX_INPROC_SERVER, &IID_IUnknown, &this->punkInside);

    if (this->punkInside)
    {
        hr = QueryInterface (this->punkInside, &IID_IFeep, &this->pfeep);
        Release (&this->ifoo);
    }
    return hr;
```

```
}
\\--------------------------------------------------------------------

static IFooVtbl vtblFoo;

static HRESULT FooQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
    else if (IsEqualIID (riid, &IID_IFeep) || IsEqualIID (riid, &IID_IBaz))
    {
//--------------------------------------------------------------------

        HRESULT hr;

        *ppv = 0;
        if (!this->punkInside && NOERROR != (hr = InitInside (this)))
                return hr;
        if (IsEqualIID (riid, &IID_IBaz))
                *ppv = &this->ibaz;
        else
                return QueryInterface (this->punkInside, riid, ppv);
\\--------------------------------------------------------------------
    }
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}
```

Tear-Off Interfaces

In some scenarios, an object may wish to support a large number of interfaces, some of which are used rarely (for example, for backward compatibility) or only transiently (for example, for initialization or in order to obtain other services, such as **IConnectionPointContainer** in the OLE Controls specification). Objects that have a lot of behavior but comparatively little state are good candidates for this technique.



A *tear-off interface* is one that depends on interface-specific reference-counting. The interface implementation is allocated by a **QueryInterface** on some other interface, and released when no one is still using the tear-off. This is a technique that really relies on clients adhering correctly to the COM rules, since releasing a tear-off really does make it go away.

An implementation of tear-offs looks a lot like the split-identity implementation shown earlier. The example below shows **COutside** rewritten so that **IBaz** is available as a tear-off.

This example shows how **CreateTearOff** delegates **IUnknown** to a **punkOuter** that is passed to it. This is done so that this same paradigm can be used when **COutside** is aggregatable, as the **IUnknown** would then need to delegate to the **punkOuter**. In this case, we simply pass in the **IUnknown** from **COutside**. For this example, we could make the tear-off four bytes smaller by accessing **COutside**'s **IUnknown** through **pThis**.

This example will create a distinct **TearOff** for each **QueryInterface** for **IBaz**. This is permitted by COM. A variant on this technique is to have the **TearOff** implement a collection of transient interfaces. If **COutside** kept a pointer to the **TearOff**, it would be possible to go to the same **TearOff** for a number of interfaces. If they are likely to be wanted together, **COutside** could keep the **TearOff** around even when its reference count is zero. It might decide to lazily deallocate it when "normal operation" is under way. This would reduce the overhead of creating multiple **TearOff**s.

This is very like delayed aggregation, except that the tear-off has access to the outer object's implementation:

```
#include "util.h"
#include "ifoo.h"
#include "ibaz.h"

typedef struct
{
    IFoo ifoo;
    int cRef;
    int value;
} COutside;

//--------------------------------------------------------------------
static HRESULT CreateTearOff (COutside *this, IUnknown *punkOuter, REFIID riid,
                              void **ppv);
\\--------------------------------------------------------------------

static IFooVtbl vtblFoo;

static HRESULT OutsideQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
    else if (IsEqualIID (riid, &IID_IBaz))
//--------------------------------------------------------------------
        return CreateTearOff (this, (IUnknown *)&this->ifoo, riid, ppv);
\\--------------------------------------------------------------------
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }

    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG OutsideAddRef (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);
    return ++this->cRef;
}

static ULONG OutsideRelease (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    if (--this->cRef == 0)
    {
        --vcObjects;

        Free (this);
        return 0;
    }
    return this->cRef;
}

static HRESULT SetValue (IFoo *pfoo, int value)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);
    this->value = value;
    return NOERROR;
}

static HRESULT GetValue (IFoo *pfoo, int *pValue)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);

    if (!pValue)
```

```
        return E_POINTER;
    *pValue = this->value;

    return NOERROR;
}

static IFooVtbl vtblFoo =
{
    (QITYPE) FooQueryInterface,
    (ARTYPE) FooAddRef,
    (RLTYPE) FooRelease,
    SetValue,
    GetValue
};


HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;

    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;

    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

    this->ifoo.lpVtbl = &vtblFoo;
    this->cRef = 1;
    this->value = 0;

    ++vcObjects;

    hr = QueryInterface (&this->ifoo, riid, ppv);

    Release (&this->ifoo);

    return hr;
}

//------------------------------------------------------------------------
typedef struct
{
    COutside *pThis;
    IBaz ibaz;
    int cRef;
    IUnknown *punkOuter;
} TearOff;

static HRESULT TearOffQueryInterface (IBaz *pbaz, REFIID riid, void **ppv)
{
    TearOff *this = IMPL (TearOff, ibaz, pbaz);
    return QueryInterface (this->punkOuter, riid, ppv);
}

static ULONG TearOffAddRef (IBaz *pbaz)
{
    TearOff *this = IMPL (TearOff, ibaz, pbaz);
    ++this->cRef;
    return AddRef (this->punkOuter);
}

static ULONG TearOffRelease (IBaz *pbaz)
{
    TearOff *this = IMPL (TearOff, ibaz, pbaz);

    Release (this->punkOuter);
    if (--this->cRef == 0)
    {

        Free (this);
```

```
        return 0;
    }
    return this->cRef;
}

static HRESULT SquareValue (IBaz *pbaz)
{
    COutside *this = SPLIT_IMPL (COutside, TearOff, ibaz, pbaz);

    this->value *= this->value;

    return NOERROR;
}

static IBazVtbl vtblBaz =
{
    (QITYPE) TearOffQueryInterface, (ARTYPE) TearOffAddRef, (RLTYPE) TearOffRelease,
    SquareValue
};

static HRESULT CreateTearOff (COutside *this, IUnknown *punkOuter, REFIID riid, void **ppv)
{
    TearOff *ptearOff;
    HRESULT hr;

    *ppv = 0;
    if (NOERROR != (hr = Alloc (sizeof (TearOff), &ptearOff)))
        return hr;
    ptearOff->pThis = this;
    ptearOff->ibaz.lpVtbl = &vtblBaz;
    ptearOff->cRef = 1;
    ptearOff->punkOuter = punkOuter;
    AddRef (punkOuter);
    *ppv = &ptearOff->ibaz;

    return hr;
}
\\-----------------------------------------------------------------------
```

Tear-Off Aggregation

If it is possible to delay aggregation to avoid the time cost of initializing an aggregate, and to tear off interfaces to avoid paying the space cost of maintaining an interface, then can we combine these two to support tear-off aggregates? The answer is Yes. The idea here is that having used an aggregated inner object for a while, it would be useful to unload the inner object before the outer object shuts down.

One might imagine that the outer object could release the **punkInner** after creating it, so that the inner would be closed down when the last use of one of its interfaces goes away. Unfortunately, this does not work. The aggregation rules require that the **IUnknown** of the inner object be kept around to keep the inner object alive. Because the inner objects are expected to simply delegate the **IUnknown** methods on other interfaces to their controlling **IUnknown**, the inner object will not be tracking the lifetime of the other interfaces. It is up to the outer object to release them first. While the inner object is free to track the reference counts and not shut down until they are all released, this is not required by the COM aggregation rules.

The problem for the outer object, then, is to know when all of the inner object's interface pointers have been released. This may be done by introducing an *intermediate controlling identity* to give to the aggregated object. The purpose of the intermediate identity is to track reference counts that occur because of delegations from the inner object. When that number reaches zero, the inner object's **IUnknown** can be released.

Here's how it's done:

```
#include "util.h"
#include "ifoo.h"
#include "inside.h"
#include "ifeep.h"

typedef struct
{
    IFoo ifoo;
```

```
    int cRef;
    int value;
//--------------------------------------------------------------------
    IUnknown iunkInner;
    int cRefInner;
\\--------------------------------------------------------------------
    IUnknown *punkInside;
} COutside;

//--------------------------------------------------------------------
static HRESULT InitInside (COutside *this, REFIID riid, void **ppv)
{
    HRESULT hr;
    if (!this->punkInside)
        hr = CoCreateInstance (&CLSID_Inside, &this->iunkInner,
            CLSCTX_INPROC_SERVER, &IID_IUnknown,
            &this->punkInside);
    if (this->punkInside)
    {
        AddRef (&this->iunkInner);
        hr = QueryInterface (this->punkInside, riid, ppv);
        Release (&this->iunkInner);
        // AR, RL shutdown punkInside if QI fails
    }
    return hr;
}
\\--------------------------------------------------------------------

static IFooVtbl vtblFoo;

static HRESULT OutsideQueryInterface (IFoo *pfoo, REFIID riid, void **ppv)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);

    if (IsEqualIID (riid, &IID_IUnknown) || IsEqualIID (riid, &IID_IFoo))
        *ppv = &this->ifoo;
//--------------------------------------------------------------------
    else if (IsEqualIID (riid, &IID_IFeep))
            return InitInside (this, riid, ppv);
\\--------------------------------------------------------------------
    else
    {
        *ppv = 0;
        return E_NOINTERFACE;
    }
    AddRef ((IUnknown *)*ppv);
    return NOERROR;
}

static ULONG OutsideAddRef (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);
    return ++this->cRef;
}

static ULONG OutsideRelease (IFoo *pfoo)
{
    COutside *this = FindImpl (pfoo, &vtblFoo);
    if (--this->cRef == 0)
    {
        Free (this);
        return 0;
    }
    return this->cRef;
}

static HRESULT SetValue (IFoo *pfoo, int value)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);
    this->value = value;
    return NOERROR;
}
```

```
static HRESULT GetValue (IFoo *pfoo, int *pValue)
{
    COutside *this = IMPL (COutside, ifoo, pfoo);

    if (!pValue)
        return E_POINTER;
    *pValue = this->value;

    return NOERROR;
}

static IFooVtbl vtblFoo =
{
    OutsideQueryInterface, OutsideAddRef, OutsideRelease,
    SetValue,
    GetValue
};

static HRESULT InnerQueryInterface (IUnknown *punk, REFIID riid, void **ppv)
{
    COutside *this = IMPL (COutside, iunkInner, punk);

    return QueryInterface (&this->ifoo, riid, ppv);
}

static ULONG InnerAddRef (IUnknown *punk)
{
    COutside *this = IMPL (COutside, iunkInner, punk);

    ++this->cRefInner;
    return AddRef (&this->ifoo);
}

//------------------------------------------------------------------
static ULONG InnerRelease (IUnknown *punk)
{
    COutside *this = IMPL (COutside, iunkInner, punk);

    if (--this->cRefInner == 0)
    {
        Release (this->punkInside);
        this->punkInside = 0;
    }
    return Release (&this->ifoo);
}

static IUnknownVtbl vtblUnknownInner =
{
    InnerQueryInterface, InnerAddRef, InnerRelease
};
\\-------------------------------------------------------------------------

HRESULT CreateOutside (IUnknown *punkOuter, REFIID riid, void **ppv)
{
    COutside *this;
    HRESULT hr;
    *ppv = 0;
    if (punkOuter)
         return CLASS_E_NOAGGREGATION;

    if (hr = Alloc (sizeof (COutside), &this))
        return hr;

    this->ifoo.lpVtbl = &vtblFoo;
    this->cRef = 1;
    this->value = 0;
//-------------------------------------------------------------------------
    this->iunkInner.lpVtbl = &vtblUnknownInner;
    this->cRefInner = 0;
    this->punkInside = 0;
\\-------------------------------------------------------------------------
```

```
    ++vcObjects;

    hr = QueryInterface (&this->ifoo, riid, ppv);
    Release (&this->ifoo);

    return hr;
}
```

Adaptive Behavior

Continuing the track of objects that scale their cost with the complexity of their use, for some objects the cost of implementing an interface may escalate dramatically for large data sets or when "high-end" features are required. At the same time, when many instances of the class are used simply, the setup time and memory cost for being prepared for the general case may be excessive.

Sometimes, the client of the object can predict easily what features will be required, and pick an appropriate class to instantiate based on its requirements. On other occasions, this is not feasible, because the pattern of usage changes during the lifetime of the object. Tombstoning is an example: An object is *tombstoned* if it represents some feature of an object that has a lifetime shorter than the reference to it. In this case, the reference has to change behavior drastically (all methods return failure).

Many programmers are familiar with the technique of using function pointers to implement such adaptive behavior: When the requirements change (for example, because the data set has reached a certain size, or some service that was being depended on has been removed), it would be useful to be able to switch the vtable of an interface, for one with an alternative "strategy."

At first sight, it would seem reasonable in these situations to have a method on an object change the **lpVtbl** pointer in one or more of its interfaces. Unfortunately, some C++ compilers, notably Microsoft Visual C++® version 2.0, cache the vtable pointers and their entries in some circumstances, so they will continue to use the old vtable after the change. Some C compilers with aggressive optimizers may even do this if the vtables are declared *const*. If the adaptive code can survive this, then changing the vtable will work, but this obviously limits the usefulness of this technique. Even if you build in C, you will still have to deal with C++ -built code in your address space. The OLE remoting infrastructure is an example. It is thus an indirect rule of COM compliance that client code is permitted to cache vtable entries, and that servers must operate correctly in the light of this.

Fortunately, when an interface is already delegating to some interface, the object can change the interface to which it delegates. If the testing required to decide on the correct behavior exceeds the cost of a delegation for an interface (about six instructions involving memory—or two to three branch tests of a member against a constant), it may be worth introducing a delegation to one of a number of specialized interface implementations. The process of choosing a class for some purpose is a way that this occurs outside an object; this form of delegation can allow it to be used within an object implementation also.

Wrap-Up

The COM standard is a fairly new technology that already has a great many uses. Hopefully, this cookbook has demonstrated that COM classes can be implemented easily and efficiently. There are a great many techniques that I have not discussed here—for lifetime management, funnels for folding similar methods together, custom handlers for optimizing cross-process performance, building cross-process code for custom interfaces, fly-weight objects, and so on. These techniques may make it into another cookbook at some future time.

COM is lightweight and adaptable. I am exploring its use as a means of structuring reusable software even within an application, as well as a means of combining software components from different sources. I think it has great potential as a robust means of building component-based software.

Happy programming!
