Bridging Object Models: The Faux-Object Idiom

Chris Sells

B.S., University of Minnesota, 1991

A thesis submitted to the faculty of the

Oregon Graduate Institute of Science and Technology

in partial fulfillment of the

requirements for the degree of

Masters of Science

in

Computer Science and Engineering

September, 1997

The thesis "Bridging Object Models: The Faux-Object Idiom" by Chris Sells has been examined and approved by the following Examination Committee:

David Maier, Thesis Adviser

Professor

Andrew Black

Professor and Department Head

James Hook

Associate Professor

Acknowledgement

I would like to thank my adviser, Prof. David Maier and my other committee members, Prof. Andrew Black and Associate Prof. James Hook, for their numerous readings and comments. The thesis would not be what it is without their involvement.

Most of all, I would like to thank my wife for her ceaseless confidence in my ability and her patience with my neglect during the long process of completing this thesis. She and my two children are what give this accomplishment meaning.

Table of Contents

List of Figures

Abstract

Microsoft's Component Object Model (COM) is the dominant object model for the Microsoft Windows family of operating systems. COM encourages each object to support several views of itself, i.e. interfaces. Each interface represents a collection of logically related functions. A COM object is not allowed to expose multiple interfaces using multiple inheritance, however, as some languages do not support it and those that do are not guaranteed to do so in a binary-compatible way. Instead, an object exposes interfaces via a function called QueryInterface(). An object implements QueryInterface() to allow a client to ask what other interfaces the object supports at run-time.

This run-time type discovery scheme has three important characteristics. One, it allows an object to add additional functionality at a later date without disturbing functionality expected by an existing client. Two, it provides for language-independent polymorphism. Any object that supports a required interface can be used in a context that expects that interface. Three, it provides an opportunity for the client to degrade gracefully should an object not support requested functionality. For example, the client may request an alternate interface, ask for guidance from the user or simply continue without the requested functionality.

COM attempts to provide its services in as efficient a means as possible. For example, when an object server shares the same address space as its client, the client calls the functions of the object directly with no third-party intervention and no more overhead than calling a virtual function in C++. However, when using COM with some programming languages, this efficiency has a price: language integration. COM does not integrate well with a close-to-the-metal language like C++. In many ways COM was designed to look and act just like C++, but C++ provides its own model of polymorphism, object lifetime control, object identity and type discovery. Of course, since C++ is not language-independent or location transparent, it was designed differently. Because of these contrasting design goals, a C++ programmer using COM often has a hard time reconciling the differences between the two object models.

To bridge the two object models, I have developed an abstraction for this purpose that I call a faux-object class. In this thesis, I illustrate the use of a specific instance of the faux-object idiom to provide an object model bridge for COM that more closely integrates with C++. By bundling several required interfaces together on the client side, a faux-object class provides the union of the operations of those interfaces, just as if we were allowed to use multiple inheritance in COM. By managing the lifetime of the COM object in the faux-object's constructor and destructor, it maps the lifetime control scheme of C++ onto COM. And by using C++ inline functions, a

faux-object can provide most of these advantages with little or no additional run-time or memory overhead.

COM provides a standard Interface Definition Language (IDL) to unambiguously describe COM interfaces. Because IDL is such a rich description language, and because faux-object classes are well defined, I was able to build a tool to automate the generation of faux-object classes for the purpose of bridging the object models of COM and C++. This tool was used to generate several faux-object classes to test the usefulness of the faux-object idiom.

Chapter 1.
Introduction

Microsoft's Component Object Model (COM) is the dominant object model for the Microsoft Windows family of operating systems. COM was developed as the architectural basis for Object Linking and Embedding (OLE). OLE is a set of communication protocols defined using COM. COM was developed for this purpose, and widely used since for many purposes besides OLE, because of several technical advantages that COM has over other object models. For example, COM provides for location transparency. A client application can be programmed for an object server that shares the same address space today and is moved to another address space, or even another machine, tomorrow. If the location of the object server changes, the same client can use the object server in its new location without a change in the source code, a re-compilation or a re-boot of the machine.

COM also provides a standard mechanism for binary compatibility between objects and clients that have been written in different programming languages or using different vendors' compilers or interpreters. A client that has been written in any language can use COM objects written in any language, so long as both languages support a COM binding[1]. This binary compatibility allows object servers to be shipped as libraries or executables, without the source code.

COM encourages each object to support several views of itself, called interfaces. Each interface represents a collection of logically related functions. A COM object is not allowed to expose an interface that has been derived from more than one interface, however, as some languages do not support it. Instead, an object exposes multiple interfaces via a function called QueryInterface(), itself part of the only required interface: IUnknown. An object implements QueryInterface() to allow a client to ask what other interfaces the object supports at run-time. This run-time type

---

[1] Of course, this would be true of any object model. The chief benefit of COM in this regard is its wide support among language and tool vendors.

discovery scheme has two important characteristics. One, it allows an object to add additional functionality at a later date without disturbing functionality expected by an existing client. Two, it provides an opportunity for the client to degrade gracefully should an object not support requested functionality. For example, the client may request an alternate interface, ask for guidance from the user or simply continue without the requested functionality.

COM provides QueryInterface() because it has no support for *type joins*. A type join is a type that is a *sub-type* of more than one *super-type*. Given a type X, a type Y is a sub-type of X if Y supports all of the operators of X and is denoted as a sub-type by the programming language in which the relationship is being defined[2]. Also, given that Y is a sub-type of X, X is defined as the super-type of Y. When this relationship is present, a routine that expects an X will work equally well with a Y because Y *conforms* to X, i.e. Y is at least everything that X is. While C++ allows type joins via multiple inheritance, COM provides no support for defining a type join. Instead, QueryInterface() provides an object's super-types, i.e. its interfaces, one at a time.

In addition to exposing object interfaces, the IUnknown interface also provides a language-independent scheme for object lifetime management, i.e. manual reference counting. Each object keeps track of its own external references via the AddRef() and Release() functions. When an interface is held by a subsystem, that subsystem lets the object know, via AddRef(), that it has one more outstanding reference. When all subsystems have released their interfaces, via Release(), the object is free to release its own resources. By maintaining the reference count in the object instead of the clients, interfaces can be passed freely between processes or machines without one client worrying when another has finished with an object.

In a distributed system, lifetime control is especially troublesome because a process on another machine or a whole machine may be stopped before it can free the object references that it holds. The COM library deals with this problem by maintaining a machine-to-machine list of object references in a list known as a *ping set*. At regular intervals (currently two minutes), a client machine will send a small data packet – a ping – to the server machine. If a certain number of pings are missed (currently three), the server will assume the client has gone down and will release the client's references automatically. This pinging mechanism is also used intra-machine so that individual client process failures can be detected without releasing all machine held object references. A server will be told of a client failure with a *delta ping*, i.e. a data packet with information about a change in the list of object references in the server's

---

[2] C++ requires explicit sub-typing using inheritance and Emerald (described in Chapter 3. Related Work) infers sub-type relationships.

ping set. This pinging mechanism is built into COM and happens without any client or object involvement and provides a reliable way for servers to be notified if clients go down without releasing outstanding object references.

So, COM provides support for language-independent, vendor-independent location transparency, run-time type discovery and lifetime control. These features are provided using interfaces as a layer of abstraction between the client and the object. COM attempts to provide these services in as efficient as possible. For example, when an object server shares the same address space as its client, the client calls the functions of the object directly with no third-party intervention and no more overhead than calling a virtual function in C++. However, when using COM with some programming languages, this efficiency has a price: language integration.

In languages that have been extended for COM, such as Visual Basic, Perl or Java, the language binding can seem to provide seamless integration with COM. COM does not integrate so well with a close-to-the-metal language like C++. In many ways COM was designed to look and act just like C++, but C++ provides its own model of object lifetime control and type discovery. C++ also provides features beyond those in COM, such as multiple inheritance and user-defined assignment and copy operations. Of course, since C++ is not language-independent or location transparent, it was designed differently (as are all language-specific object models, e.g. Java). Because of these contrasting design goals, a C++ programmer using COM often has a hard time reconciling the differences between the two object models.

Fortunately, C++ provides the ability to wrap the abstractions of COM into classes that integrate more closely with the language. I have developed an abstraction for this purpose that I call a faux-object class. Its job is to provide a bridge between two different object models. In this thesis, I use the faux-object idiom to provide an object model bridge for COM that more closely integrates with C++. By bundling several required interfaces together on the client side, a faux-object class provides the union of the operations of those interfaces, just as if we were allowed to use multiple inheritance in COM. In affect, the faux-object is providing the type join for C++ that COM lacks. Also, by managing the lifetime of the COM object in the faux-object's constructor and destructor, the faux-object maps the lifetime control scheme of C++ onto COM. By implementing a copy constructor and assignment operator using a standard COM persistence interface, a faux-object class can provide C++ copy and assignment semantics for those COM objects that implement that interface. And by using C++ inline functions, a faux-object can provide most of these advantages with little or no additional run-time or memory overhead.

Finally, COM provides a standard Interface Definition Language (IDL) to unambiguously describe COM interfaces. IDL is an extended version of the Open

Software Foundation's Distributed Computing Environment Remote Procedure Call IDL. The COM version of IDL is a superset of this industry standard that has been extended to define interfaces. Because IDL is such a rich description language, and because faux-object classes are well defined, I was able to build a tool to automate the generation of faux-object classes for the purpose of bridging the object models of COM and C++. This tool was used to generate several faux-object classes to test the usefulness of the faux-object idiom. As its input, the tool uses standard Microsoft Interface Definition Language (IDL) files.

This thesis is organized into several chapters. Chapter 1 is this introduction. Chapter 2 will describe the major components of COM and how it integrates with C++. Chapter 3 will describe related work. Chapter 4 will preset the faux-object idiom and how it is used to provide a bridge between the COM and the C++ object models. Chapter 5 will describe the faux-object class generation tool and show some simple examples. Chapter 6 will discuss the introduction of a generated faux-object class in a body of existing code to replace the use of raw COM interfaces. Chapter 7 is a discussion of how well the faux-object idiom met its goals and how it can be extended in the future. Chapter 8 is a list of references. Appendices A through F are a set of code examples used to support points made in the thesis.

Chapter 2.
The Component Object Model

2.1 COM Interfaces

The central concept of COM [COM95] is the separation of interface from implementation. A COM *implementation* (also called a COM class) is a black box of behavior and state to a COM client. The only way for a client to access the functionality of a COM implementation is via one or more COM *interfaces* (an implementation will normally support several interfaces). A COM interface is three things:

- A collection of logically related member functions.
- An immutable physical layout.
- An optional packet format for passing member function arguments between processes and machines.

To avoid confusion, when referring to the physical layout of an interface, I'll use the term *interface layout* and when referring to the packet format, I'll use the term *interface packet format*.

For example, the following is the definition of an interface that represents operations on a string:

```
interface[3] IString : public IUnknown[4]
{
    // IString member functions inherited from IUnknown
    HRESULT QueryInterface(REFIID riid, void** ppv) =0;
    ULONG   AddRef() =0;
    ULONG   Release() =0;
```

[3] The 'interface' keyword is just a type definition for the C/C++ keyword 'struct.'

[4] All interfaces must derived from IUnknown.

```
    // IString-specific members
    HRESULT SetText(const char* szText) =0;
    HRESULT GetText(char** ppszText) =0;
    HRESULT GetLength(int* pnLength) =0;
};
```

This interface would correspond to the following physical interface layout:

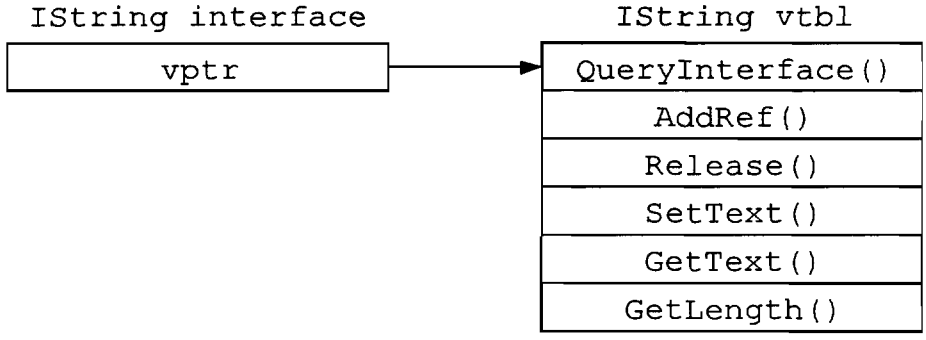


Figure 1: IString interface layout

The physical layout of an interface must remain unchanged once it has been published. Compiled clients rely on the *virtual function table* (vtbl) layout to perform vtbl-binding at compile-time. This form of binding is very efficient, but relies on the physical layout of an interface to remain unchanged between the time the client is compiled and the interface is actually used.

The packet format for the IString interface would describe how to properly *marshal* the member function parameters between processes or machines, e.g., copy parameters between one address space and another. Marshalling parameters is necessary because objects are not typically passed by value in COM, but by reference. An interface pointer is a reference to one of the base classes that an object implements. When calling interface member functions, the client often references an object that exists in a separate address space or on a different machine than the client. For the object to perform the requested operation, the parameters must be marshaled from the client's address space into its own.

I should mention that while objects are most often passed by reference in COM (via interface pointers), other member function parameters are going to be copied from one address space to another for use by the object's implementation. The process of serializing the parameters of a member function call from the address space of the client to that of the object is performed by a helper object known as a *proxy*. It's the proxy's job to pretend to be the object for the client, but to bundle up the parameters that the object needs to provide its implementation, i.e. *in parameters*, and communicate them to a helper object in the object's address space. This other helper

object is called a *stub*, and its job is to unpack the serialized parameters, push them onto the stack and call the proper member function of the actual COM object. Any parameters that may have been updated by the object's member function implementation, i.e. *out parameters*, need to be copied back into the client's address space at the completion of the member function call. The idea is that both the client and object can pretend to be in the same address space and the proxy and stub use marshalling to maintain this illusion.

2.2 IUnknown

Every interface must ultimately *derive* from the universal COM base interface: IUnknown, i.e. every interface is a sub-type[5] of IUnknown. IUnknown provides two services: run-time type discovery and lifetime control. The IUnknown interface is defined as follows:

```
interface IUnknown
{
    HRESULT QueryInterface(REFIID riid, void** ppv) =0;
    ULONG   AddRef() =0;
    ULONG   Release() =0;
};
```

The QueryInterface() member function is used by the client to request an interface from an object. For this purpose, QueryInterface() takes a 128-bit number that uniquely identifies the interface called an *interface identifier* (IID). The COM sub-system provides a routine to generate unique identifiers called *Globally Unique Identifiers* (GUIDs). IIDs are instances of GUIDs used with QueryInterface(). QueryInterface() uses IIDs to provide a safe run-time type discovery mechanism conceptually identical to the C++ dynamic_cast operation. Both QueryInterface() and dynamic_cast allow for a client to take a reference to an object and ask if it supports functionality other than that expressed in the current reference type. The implementation of dynamic_cast is C++ vendor-specific while QueryInterface() works across vendors and languages.

Instead of relying on a single client to define the lifetime of an object using automatic scoping (object is allocated from the stack) or manual scoping operations (object is allocated from the heap), COM objects use a reference counting model. This model allows interfaces to be passed between subsystems without regard for the normal C++ convention of "whoever creates an object must free it." The C++ convention works fine (mostly) when a single individual or group controls all of the code that

---

[5] The relationship of COM and types is described in Chapter 3, Related Works.

references an object. However, in the distributed world, object references are passed between subsystems, processes and machines. Instead of making the object-creating client stick around until everyone on the planet is finished with the object, the client simply manages the lifetime of the object using the interfaces it holds (via the AddRef() and Release() member functions present in every interface) and lets other clients of the object do the same. The object itself maintains its own lifetime count and releases its resources when it feels free to do so, e.g., when all clients have released all outstanding interfaces.

2.3 COM Implementations

A COM class bundles one or more interfaces together. A COM class is uniquely identified via another GUID called a *class identifier* (CLSID) to allow a client to request an object that provides an implementation of a set of interfaces. However, the interfaces that a class supports cannot be known until runtime. By allowing the implementation to be changed, clients get objects that evolve. Further, by fixing the interface layout, implementations can evolve safely. Object evolution, or versioning, is currently a problem with language or vendor-specific object models, which typically provide syntactic separation of interface and implementation only. Because the separation is blurred at the binary level, clients have intimate knowledge of the implementation of an object. For example, for a client to be able to access a C++ object that exists in a *Dynamic Link Library* (DLL)[6], the client and the DLL must agree on implementation details like object layout, space requirements and symbol decoration conventions[7] [Lippman96]. This agreement can typically only be achieved if the client and the object are developed using the same language, the same compiler vendor and the same compiler version. Further, even if the client and the DLL do agree on implementation details, any change in the object layout or space requirements as the object implementation evolves requires the client to recompile. All of these requirements are lifted using a binary separation, rather than a syntactic separation, of interface and implementation such as the one that COM specifies.

2.4 COM/C++ Integration

It could be argued that the reason that COM has QueryInterface() is because interfaces may only derive from a single base interface. If COM interfaces could derive

---

[6] DLL is a Windows term for a library of code that is loaded into the address space of the client as needed.

[7] Most C++ implementations use a technique called *symbol decoration* to implement type safe linkage. Unfortunately, all vendors do it their own way.

from multiple base interfaces, the most derived interface on an object would expose all of the public functionality of the object, thus removing a great deal of the need for QueryInterface(). For example, CORBA [Siegel96] provides a distributed object model similar to COM, but it supports interfaces with multiple bases. ORB vendors provide tools to generate a language binding that closely integrates with the language of interest. Those languages that do not support the features of CORBA directly get a generated wrapper that provides the functionality in a language-intelligent way.

Instead of providing tools to generate language bindings to map the object model of COM onto the object model of a specific language, COM defines only the required interface layout. Binary compatibility provides several advantages, but it also yields somewhat of a lowest common denominator approach. The only features available are the features that can readily be mapped into all languages in a way that is binary compatible. The lack of language-specific bindings relieves Microsoft of the chore of building a code generator for every language that comes along, but it also requires a developer to understand how their language maps to the interface layout requirements of COM. Achieving this binary compatibility sometimes requires the use coding conventions that are very different from the conventions typically used with the language. For example, in C++, a C++ object can be created using the new operator, but a COM object will most often be created using the COM function CoCreateInstance(). Understanding the need for these two different methods for obtaining an object in the same language is often daunting and sometimes prohibitive.

This issue gets at the root of the difference between CORBA and COM. CORBA chose language integration over performance and COM went the other way. Because multiple-inheritance is not supported in many languages, member function calls under CORBA take longer to execute. The extra time is needed for the function call on the interface to be mapped to the function call on the implementation. Under COM, no mapping is required. An interface member function call to an object in the same address space[8] incurs no more overhead than a C++ virtual member function call.

It worth noting that this performance comparison assumes an object that lives in the same address space as the caller. While both COM and CORBA must have some part of an object in the same address space as the client, when the in-process part is merely the proxy for an object in another address space, the function call overhead is insignificant when compared to that of marshaling the parameters to the stub in the object's address space and switching control. Herein lies another difference between

---

[8] This is not always true. Sometimes calls between threads result in extra overhead to perform concurrency control for objects that would not otherwise be thread-safe.

CORBA. CORBA was designed to handle distributed objects and not all ORBs implement same address space objects at all. COM was designed first for objects in the same process and on the same machine and later extended for objects on other machines. This trade-off is because the original operating system that COM was designed to work under – Windows – is based on DLLs. Fortunately, the model that COM uses scales well to distributed objects.

While the binary separation of interface and implementation provides performance benefits, there are advantages to the model that CORBA uses. One benefit is that CORBA provides a rich object model very close to that of C++. For languages that do not support all of the features of C++, e.g., multiple-inheritance under C, CORBA provides a language mapping to simulate these features. COM only uses a single language feature: pointers[9]. A vtbl is just a table of function pointers. Even an assembly language programmer can program to an object model expressed in those terms. However, for the C++ programmer expecting an object model that handles multiple inheritance and automatic object scoping, the features of the COM model can seem primitive and complicated.

The problem has to do with the very thing that makes COM so powerful – the separation of interface and implementation. Under C++, a programmer is accustomed to defining a class that derives from any number of base classes. A successful creation of an object at run-time guarantees the ability to call any of the member functions of the object's class, including any of the members of the base classes. I call this ability *implementation guarantee*. By compiling a member function call, the compiler[10] guarantees the implementation of that member function will be there at run-time. For example, here's the definition and use of a simple C++ class:

```
#include <iostream.h>
#include <string.h>

class String
{
public:
    String() : m_sz(0) {}
    ~String() { free(m_sz); }
```

---

[9] Languages that do not support pointers must have support for COM added to them. Only languages that support pointers, and pointers to functions, can using COM interfaces without explicit support for COM.

[10] For simplicity, I'll refer to the operations of the compiler and linker together as operations of the compiler.

```
    void        SetText(const char* sz) { m_sz = strdup(sz); }
    const char* GetText() { return m_sz; }
    int         GetLength() { return strlen(m_sz); }

private:
    char*   m_sz;
};

void main()
{
    String  s;
    s.SetText("Hello, World");
    const char* psz = s.GetText();
    long        nLen = s.GetLength();
    cout << psz << " (" << nLen << ")" << endl;
}
```

If the compiler allows this code, the constructor, destructor, GetText(), SetText() and GetLength() member functions are guaranteed to be implemented at run-time. The compiler has complete knowledge and can check, at compile-time, whether all needed functions are implemented or not. The client of a C++ object can simply create an object and assume the implementation is available, because it is guaranteed to be.

On the other had, the client of a COM object does not have this guarantee. The compiler cannot guarantee the availability of the implementation because the implementation is not chosen until run-time. The burden is on the client, therefore, to check for desired functionality before it can be used. This forces the client to deal with a whole new class of errors, i.e., what to do if that functionality is not available. Here is an example of a COM client creating and using a simple COM object:

```
#include <windows.h>
#include <iostream.h>

// This IID uniquely identifies the IString interface
const IID IID_IString =
    {0x73F86A20,0x621C,0x11cf,
        {0x88,0xD2,0x00,0x00,0x86,0x00,0xA1,0x05}};

// This CLSID uniquely identifies the CoString
// implementation
const CLSID CLSID_CoString =
    {0x0845D620,0x621A,0x11cf,
        {0x88,0xD2,0x00,0x00,0x86,0x00,0xA1,0x05}};
```

```
// This defines the IString interface for client use
interface IString : public IUnknown
{
public:
    virtual HRESULT STDMETHODCALLTYPE
        SetText(const char* szText) = 0;

    virtual HRESULT STDMETHODCALLTYPE
        GetText(char** pszText) = 0;

    virtual HRESULT STDMETHODCALLTYPE
        GetLength(long* pnLen) = 0;
};

void main()
{
    HRESULT hr;
    CoInitialize(0);

    // Create an object of type CoString
    IUnknown*   punk;
    hr = CoCreateInstance(CLSID_CoString, 0,
                          CLSCTX_ALL,
                          IID_IUnknown,
                          (void**)&punk));
    if( SUCCEEDED(hr) )
    {
        // Ask the object for the IString interface
        IString*    ps;
        hr = punk->QueryInterface(IID_IString,
                                  (void**)&ps)
        if( SUCCEEDED(hr) )
        {
            // Ask the object for the IPersist interface
            // (a standard interface defined in windows.h)
            IPersist*    pp;
            hr = punk->QueryInterface(IID_IPersist,
                                      (void**)&pp)
            if( SUCCEEDED(hr) )
            {
                // Safe to use IString and IPersist
                char*   psz = 0;
                long    nLen = 0;

                ps->SetText("Hello, World");
                ps->GetText(&psz);
```

```
                ps->GetLength(&nLen);

                wchar_t* pszClsid;
                CLSID   clsid;
                pp->GetClassID(&clsid);
                StringFromCLSID(clsid, &pszClsid);

                cout << psz << " (" << nLen << ")"
                     << " from " << szClsid
                     << endl;

                // Release resources client has acquired
                CoTaskMemFree(psz);
                CoTaskMemFree(pszClsid);
                pp->Release();
            }
            ps->Release();
        }
        punk->Release();
    }
    CoUninitialize();
}
```

In this example, the COM client first creates an instance of the COM class uniquely identified by the CLSID_CoString GUID by calling CoCreateInstance(). Part of the creation process is asking for the initial interface from the object. Since the client is never allowed access to the implementation directly, it must pick an interface that it would like to access initially (in this case, IUnknown). From the initial interface, all other interfaces required by the client must be obtained via QueryInterface().

If an object of the requested type is available (it might not be), the client then asks for the "string" functionality of the object by querying for the availability of the IString interface by calling the QueryInterface() member function using the IID_IString GUID. If that functionality is available (again, it might not be), the client is allowed to use it. To use another interface, i.e., IPersist, the client must call QueryInterface() again. The IPersist interface is a standard interface defined by Microsoft and it provides a single member function: GetClassID().

To access all of the required functionality, the client must manage three separate interface pointers. For each interface that the client has acquired, a matching Release() call must be made. The QueryInterface() member function does an implicit AddRef() on every interface successfully returned.

As you can see, while the functionality provided by the C++ string object and the COM string object are identical (with the exception of the COM-specific

functionality exposed by the IPersist interface), the client code to use the COM object is quite a bit more complicated. The client is required to manage separate interfaces to access functionality, to manually control the object's lifetime and to check for implementation guarantees at run-time. In general, the differences in the object models makes it difficult for a C++ programmers to enjoy of the benefits of COM.

Luckily, C++ has a built-in mechanism to extend the functionality of the language: classes. It is possible to build a C++ class to map the implementation guarantee, multiple-inheritance and lifetime control mechanisms of C++ onto COM. This mapping uses the facilities of C++ to build a CORBA-like language mapping onto COM, thereby merging the benefits of the three object models – C++, CORBA and COM. The provision of this mapping is the subject of this thesis.

Chapter 3.
Related Work

3.1 Distributed Object Models

The Eden [Almes85] system provided the foundation for many distributed object systems and includes support for concurrency, transactions and persistence. Eden is not currently used.

The Emerald [Black86, Black87] system is an object-based language that supports distributed objects. What makes Emerald especially interesting is its support for types. An Emerald object's type describes methods it supports and the types of the arguments and results of those methods. An identifier of a given type can be bound to any objects that support the required methods. For example, given a type X, a type Y is implicitly a sub-type of X if it supports all of the operations of X and takes the same number and types of arguments. Likewise, in this example, X is a super-type of Y. Given this implicit sub-typing system, an operation can define its own types of arguments, based on its requirements, and the compiler will determine if the passed in argument types *conform* to the required argument types.

The Argus [Liskov88] system is a distributed object system with its own programming language, also called Argus. It is designed to handle the special problems of distributed systems, e.g., concurrent processes, dropped connections and parts of the system that have crashed. It does this by providing support for transactions, dynamic load distribution and replication. The Arjuna [Shriv91] system is based on this work and offers similar benefits. However, it provides a C++ language mapping instead of requiring its own language.

The Advanced Network Systems Architecture (ANSA) Computational Model [APM91] builds on the work of Emerald and Argus in an attempt to define a more robust model of distribution, concurrency and heterogeneity. It defines these key concepts:

Object: a unit of program modularity having state and operations.

Interface: a view of an object as an abstract service specified as a set of operations.

Operation: a part of an interface having a signature and a body, defining the effect of an invocation of the operation.

Signature: the name of an operation, the number and types of the arguments.

Interface Type: a schema for an interface, the signatures of the operations in interfaces of the type.

Invocation: the execution of the body of an operation defined by a reference to an interface and an operation name in a context established by the referenced interface and a set of arguments.

Server: in the context of an invocation, the object that provides the interface containing operation being invoked.

Client: in the context of an invocation, the object from which the invocation was initiated.

While ANSA-specific products have never achieved commercial success, the conceptual model forms the basis for the distributed systems to follow.

Smalltalk [LaLonde90] is an object-oriented language that has had distributed implementations [Bennett90, LaLonde90a, Keremitsis95]. Smalltalk is based on the idea that everything is an object and communication happens by sending a message from one object to another. However, the lack of type checking and the high machine requirements have kept it from being widely adopted.

Java [Flanagan96] is a recent entry into the world of object-oriented languages. It is based on many of the same principles as Smalltalk, but it is strongly typed. In addition, objects that a running Java application can support can be augmented at runtime by downloading code from the Internet. With the recent addition of Java Remote Method Invocation (RMI) [RMI96], Java provides for distributed method invocations. The chief downside of Java is that clients and object may only be developed in Java. This will change, however, when Java supports calling objects via the Internet Interoperability Protocol (IIOP) [Curtis97], as was recently announced. Current, Java does have some support for invocation of functions exposed from a C library, but this is operating system dependent.

IIOP is part of the Common Object Request Broker Architecture (CORBA) specification and allows Object Request Broker (ORB) implementations from multiple vendors to communication with each other. CORBA [Siegel96] is unquestionably the most popular, widely supported distributed object model is use today. This operating system-neutral, language-neutral, vendor-neutral standard was designed to distribute

objects across process and machine boundaries. It provides a super-set approach; i.e., it supports features (such as sub-typing) that some languages do not support. The language mappings for these languages provide an implementation of any missing features.

COM [COM95] takes a different approach to distributed objects than CORBA. While COM has always supported communication between objects in different address spaces on the same machine, it was designed to be extremely efficient when objects share the same address space. It is only recently that this model has been extended to support objects that live across the machine boundary. This support is based on the Open Software Foundation's Distributed Computing Environment Remote Procedure Call (OSF DCE RPC) specification [RPC93]. Microsoft has provided an implementation of DCE RPC that extends the wire representation and the interface definition language to support COM interfaces. In fact, COM is often referred to as "RPC with a *this* pointer." The "this" pointer is what gives COM its object-oriented programming model. RPC itself is strictly procedural.

COM leverages the ideas defined by ANSA, including the ANSA idea that an object provides multiple interfaces. However, there is one major difference – typing. The ANSA model allows an interface to be a sub-type of more than one super-type and the type relationship is inferred. COM, on the other hand, requires an interface to be a sub-type of exactly one super-type, with IUnknown being the top-most super-type. COM allows the language vendor to determine whether sub-typing happens explicitly, implicitly or not at all. For example, C++ requires explicit sub-typing while C has no notion of type conformance of any kind. For objects that support multiple interfaces, all interfaces provide the QueryInterface() operation, which acts as a language-independent mechanism for obtaining a class's super-types, even in languages that do not support sub-typing. Unfortunately, in COM, this mechanism is also required for languages that do support sub-typing because COM does not define a join of the super-types supported by the object. Instead, COM uses QueryInterface() to provide one super-type at a time.

3.2 COM Language Bindings

Since version 4.0, the Visual Basic language [VB97] has provided a built-in mapping between Visual Basic (VB) and COM. It can do this easily because Microsoft controls the specification and implementation of the language. Modern implementations of VB support interfaces and implementations using the COM object model. All type coercion and lifetime control is done using IUnknown. In short, in VB there is no mapping between VB and COM – VB is COM.

Microsoft's implementation of Java [Java96], on the other hand, does require some language mapping to support COM. Microsoft's Java Virtual Machine (VM) implementation supports dynamic binding to Java classes and COM classes. To facilitate this process, Microsoft provides a tool that takes a description of one or more COM interfaces and produces a Java interface for each one. A COM object can be created using the Java new operator, which has been extended to understand when it is to use the COM function CoCreateInstance(). Once the object has been created, the Java casting syntax can be used to perform a QueryInterface() to access the interfaces provided by the object. A successful QI() yields a valid interface reference while a failed QI() produces a Java exception. These extensions to Java support COM in a way that closely integrates with the language. Unfortunately, it is a feature unique to Microsoft's VM implementation and thus far has not been popular. Not only is it platform-specific, but it is specific to one vendor's VM implementation and therefore goes against the very nature of Java.

The C++ language binding for COM is, like Visual Basic's, a thin one and unlike Java's, does not require any changes in the implementation of the language itself. Once a reference to an interface has been obtained, it can be treated like a pointer to one of an object's super-types. However, the design goals of COM clash with those of C++, providing a gap that the developer must bridge. This gap includes differences in object lifetime and interface management. The faux-object idiom described in this thesis is presented to bridge this gap for differing object models in general and COM and C++ specifically.

Chapter 4.
Faux-Object Idiom

The goal of a faux-object is to provide a mapping between the C++ and the COM object models. This reduces coding complexity for the client while still taking advantage of objects implemented by the server. In general, the mapping will entail accessing object functionality and managing object lifetime, although every object model pair will have its own mapping requirements. Specifically, the faux-object to map the client-side C++ object model onto the server-side COM object model has the following requirements:

- **Join multiple COM interfaces together into a single C++ object.** A C++ programmer is used to accessing all of the functionality of an object via a single interface. COM programmers, on the other hand, are forced to deal with separate sets of functionality (interfaces) all implemented by the same object. By joining multiple COM interfaces into a single C++ class, a C++ client can simply call member functions and have them mapped to the appropriate COM interface member function.
- **Translate C++ object scoping rules into COM reference counting rules.** The reference counting model of COM is extremely useful for centralizing resource management in the object implementation. However, proper reference counting is yet another detail a C++ programmer must deal with when using COM interfaces. A faux-object will be created using standard C++ scoping and the reference counting model of COM will be handled transparently by the faux-object.
- **Provide an implementation guarantee for client-required functionality.** The C++ compiler can guarantee that an object implements all of its member functions. On the other hand, the C++ client of a COM object must check for functionality via QueryInterface() at runtime. A faux-object class can provide a guarantee of required interfaces at runtime just like the C++ compiler does at compile time.

- **Provide C++ type compatibility with individual interfaces.** Many of the functions exposed by existing COM and OLE libraries require COM interface pointers as arguments. To allow clients of faux-objects to take advantage of this base of code, as well as any existing COM-based client code, a faux-object allows direct access to any of the interfaces it supports. Direct interface access allows seamless use of a faux-object as a function argument where a COM interface pointer would normally be used.

How the faux-object idiom meets these requirements is discussed in detail below. Meeting these requirements will allow a C++ client program to treat a COM object like any other C++ object without losing the added functionality provided by COM. The pattern of implementation that I will present meets these goals for any collection of COM interfaces implemented by a COM object. This pattern represents the faux-object idiom.

The faux-object was named for several reasons. Most importantly, the faux-object is really just a "fake" object that wraps one or more interfaces implemented by a COM object. Nearly all of the interface member functions of the faux-object are simple pass-throughs to the underlying COM object implementation. Instead of using the COM interfaces directly, the C++ client uses an instance of the faux-object to hide the underlying COM complexity.

Secondarily, server-side COM implementations are generally named with the "Co" prefix to mean Component Object, e.g., CoString. Likewise, the faux-object implementation will be entirely part of the client code and the names will begin with the "Fo" prefix to mean Fake Object, e.g., FoString. Finally, the fact that "Co," "Fo" and faux all rhyme provides the third leg that stabilizes my name choice.

4.1 Reduced Complexity

The following is the partial definition of a faux-object class to join the three interfaces required by the previous COM client, i.e., IUnknown, IString and IPersist:

```
class FoString
{
public:

// Constructor and Destructor
    FoString(REFCLSID clsid, DWORD ctx = CLSCTX_ALL[11]);
```

[11] The Class Context (CLSCTX) allows the client to specify acceptable locations for an object to be loaded, i.e., in-process or out-of-process. CLSCTX_ALL means that the client does not care where the object is loaded.

```
    virtual ~FoString();

// IUnknown Pass-Throughs
    // AddRef() and Release() not needed for C++ scoping
    HRESULT QueryInterface(REFIID riid, void** ppv);

// IString Pass-Throughs
    HRESULT SetText(LPSTR szText);
    HRESULT GetText(LPSTR* pszText);
    HRESULT GetLength(long* pnLen);

// IPersist Pass-Throughs
    HRESULT GetClassID(struct _GUID* pClassID);
};
```

By using a faux-object, a C++ client using a COM object can gain a substantial reduction in coding complexity. The following is an example usage of the faux-object defined above:

```
#include <windows.h>
#include <iostream.h>

// COM GUIDs and interface definitions
// still required but removed for clarity

// Define the FoString that joins IString and IPersist
#include "FoString.h"

void main()
{
    CoInitialize(0);

    try
    {
        // Bind the faux-object to an implementation
        FoString    s(CLSID_CoString);

        // Use IString members in FoString
        char*   psz = 0;
        long    nLen = 0;

        s.SetText("Hello, World");
        s.GetText(&psz);
        s.GetLength(&nLen);

        wchar_t* pszClsid;
```

```
        CLSID    clsid;
        s.GetClassID(&clsid);
        StringFromCLSID(clsid, &pszClsid);

        cout << psz << " (" << nLen << ")"
             << " from " << pszClsid
             << endl;

        // Release resources client has acquired
        CoTaskMemFree(psz);
        CoTaskMemFree(pszClsid);
    }
    catch( ... )
    {
        cerr << "Unable to create CLSID_CoString.\n";
    }

    CoUninitialize();
}
```

Notice how the object creation is wrapped in a try-catch block. Because COM object implementations are bound at run-time (instead of compile-time, as in C++), it is possible that the requested implementation is not available. If the implementation is available, however, the constructor caches all of the required interfaces. Should the implementation or any of the required interfaces be unavailable, an exception will be thrown. Otherwise, if the member functions can be called, their implementations are guaranteed to be available. This gives an implementation guarantee very much like that provided by C++.

Secondly, notice that the client uses the functionality provided in two separate interfaces, the IString interface defined above and the IPersist interface. In the example above, the client can simply call the GetClassID() member function without additional interface management because the faux-object provides support for both the IString and the IPersist interfaces.

Finally, notice the lifetime control of the faux-object. Instead of any calls to release acquired interfaces, the client simply lets the faux-object go out of scope. When this happens, the faux-object destructor will be called and it will release the cached interface references.

This sample shows how the complexity of COM, e.g., manual implementation guarantee, interface management and manual lifetime control, can be avoided without giving up the benefits of COM, e.g., run-time binding of implementation and location independence.

4.2 Faux-Object Implementation

This section describes each of the features provided by the faux-object and the way in which those features are implemented. The basic implementation philosophy is to provide a single class, e.g., FoString, that caches each of the interfaces required by the client at construction time. It also provides a union of all of the member functions of all of the supported interfaces. To provide the functionality of the underlying object to the client, the implementation of each of the faux-object member functions simply calls the member function of the appropriate cached interface. At destruction time, each cached interface is released.

The following is a list of the requirements of a C++/COM faux-object class implementation and how each requirement is met.

4.2.1 Join multiple COM interfaces together into a single C++ object

When an object is created in C++, the client of the object knows its class, which defines the functionality of the object. A C++ object's functionality is the union of all of the member functions in all of the base classes plus those of the derived class. For example, consider a C++ Mule object that derives its interface and implementation from its base classes, Horse and Donkey. In C++, Donkey, Horse and Mule could be defined like this:

```
class Donkey
{
public:
    virtual void Bray();
};

class Horse
{
public:
    virtual void Prance();
};

class Mule : public Donkey, public Horse
{
};
```

The resulting class hierarchy is shown in Figure 2.

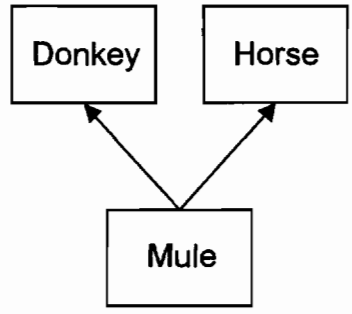


Figure 2: Mule class inheriting from both Donkey and Horse classes

A C++ client could create an instance of a Mule object and have access to any member function with equal notational convenience regardless of the base class of which it is a member.

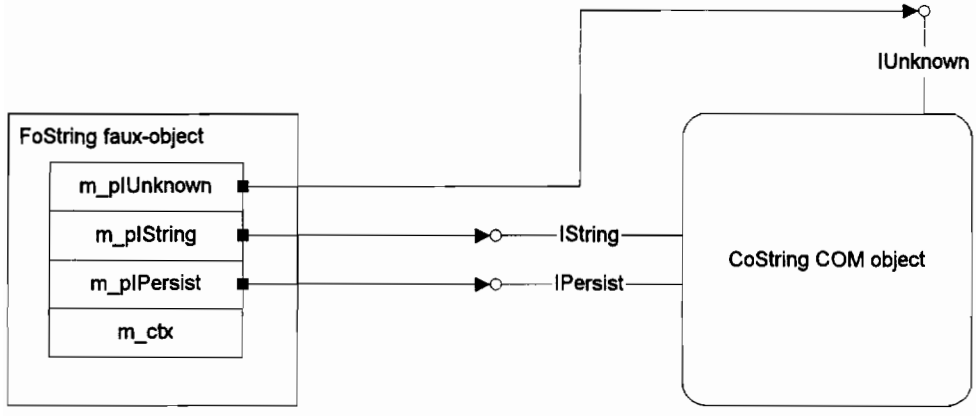
On the other hand, in COM, the Mule could not be an interface exposed directly by an implementation because it has more than one super-type. Instead, the Mule would be an implementation which would expose two separate interfaces, Donkey and Horse. The inability to define the Mule as the union of all of the member functions of the base classes removes the notational convenience provided by a C++ object: the client must get a Horse interface reference to access Horse functionality and a Donkey interface reference to access Donkey functionality. Forcing the use of separate interfaces robs the C++ client of the notational convenience it expects when using an object.

To simulate the notational convenience of a type with multiple super-types, a faux-object manually joins together the interface member functions of all of the supported interfaces. The implementation of each of these member functions is a *pass-through* to the underlying object via the appropriate interface. For example, the FoString faux-object supports three interfaces and implements the pass-through member functions like this:

```
class FoString
{
// IUnknown Pass-Throughs
    // AddRef() and Release() not needed for C++ scoping
    HRESULT QueryInterface(REFIID riid, void** ppv)
    { return m_pIUnknown->QueryInterface(riid, ppv); }

// IString Pass-Throughs
public:
    HRESULT SetText(LPSTR szText)
    { return m_pIString->SetText(szText); }

    HRESULT GetText(LPSTR* pszText)
    { return m_pIString->GetText(pszText); }
```

```
    HRESULT GetLength(long* pnLen)
    { return m_pIString->GetLength(pnLen); }

// IPersist Pass-Throughs
public:
    HRESULT GetClassID(struct _GUID* pClassID)
    { return m_pIPersist->GetClassID(pClassID); }

// Other members removed for clarity
};
```

An instance of FoString would be represented in memory as shown in Figure 3 (the m_ctx member is used to cache the context that the object is created in, i.e. in the



same address space, on the same machine or on another machine).

Figure 3: Faux-object memory layout

Because the faux-object has provided an implementation guarantee, the pass-through member functions can blindly forward the calls to the appropriate interface member function. In this way, the client can simply call the appropriate member function without the overhead of an additional QueryInterface()/Release() pair or using separately cached interfaces based on the functionality required. Further, letting the faux-object manage the interfaces internally eliminates the confusion that often arises from having multiple interfaces to the same object.

4.2.2 Translate C++ object lifetime rules into COM reference counting rules

The C+ + object model uses a block structure that allows objects to be created from two *object stores,* i.e., the stack and the heap. If an object is automatically allocated on the stack, when the name of the object goes out of scope, the object will be destroyed. In an object is manually allocated on the heap, the scope of the name of

the object does not affect the lifetime of the object. Instead, the programmer is responsible for determining when the object should be destroyed.

In contrast, COM requires the use of manual reference counting. Each time a reference to an object is created, the object is notified via a call to the AddRef() member function of the IUnknown interface. Likewise, when a reference is destroyed, a call to Release() is made. The lifetime of a COM object is determined by all of the outstanding references held by all clients of the object. Once all references have been released, the object is free to release its own resources.

The benefit of COM reference counting is that many parts of a distributed system can hold onto a reference to an object without concern for any other outstanding references. Instead of a client mistakenly destroying an object that another client relies on, a client destroys its reference and lets the object worry about the total number of references. The downside of this technique is that every client is forced to implement the rules of COM reference counting properly or an object will not be able to manage its lifetime accurately.

By using a faux-object, the C++ programmer is freed from the responsibility of manual reference counting and may allocate the faux-object from either the stack or the heap. Whatever is used, the object's *constructor* is called when it is created and its *destructor* is called when the faux-object is destroyed (either automatically or manually via `delete`). The constructor and the destructor are the object's opportunity to properly initialize the object and release any resources held by the object respectively. In the faux-object, the constructor creates an instance of the underlying COM object specified by an argument to the constructor and caches all of the required interfaces. Likewise, the faux-object destructor releases all of the cached interfaces.

The faux-object has two constructors that create a COM object. One takes a CLSID and one takes a *programmatic identifier (ProgID)*. A programmatic identifier is a string-based identifier that can be mapped to a CLSID using a machine-wide database. This ProgID constructor is provided for convenience only as it is functionally equivalent the CLSID constructor. The following is the implementation of the two faux-object constructors that create a COM object. Note that both of them call a helper function discussed below:

```
class FoString
{
// GUID Constructor
public:
    FoString(REFCLSID clsid, DWORD ctx = CLSCTX_ALL)
    { Create(clsid, ctx); }

// ProgID Constructor
```

```
public:
    FoString(LPOLESTR szProgID, DWORD ctx = CLSCTX_ALL)
    {
        CLSID clsid;
        if( SUCCEEDED(CLSIDFromProgID(szProgID, &clsid)) )
        {
            Create(clsid, ctx);
        }
        else
        {
            throw XCreateFoStringException();
        }
    }

// Other members removed for clarity
};
```

The following is the implementation of the Create() member function called by the constructors of the FoString class:

```
void FoString::Create(REFCLSID clsid, DWORD ctx)
{
    // Define a table of required interfaces
    // for this particular faux-object class
    MULTI_QI aqi[3] = {
                        {&IID_IUnknown, 0, 0},
                        {&IID_IString, 0, 0},
                        {&IID_IPersist, 0, 0},
                    };

    // If creation succeeded and all interfaces returned,
    if( CoCreateInstanceEx[12](clsid, 0, ctx,
                         0, 3, aqi) == S_OK )
    {
        // Cache all interface references from the table
        m_pIUnknown = (IUnknown*)aqi[0].pItf;
        m_pIString = (IString*)aqi[1].pItf;
        m_pIPersist = (IPersist*)aqi[2].pItf;
    }
    // If creation failed to return any or all interfaces,
    else
    {
```

[12] The use of CoCreateInstanceEx() in the previously shown example COM client would not have freed the client from managing separate interfaces.

```
            // Release the interfaces returned
            for( int i = 0; i < 3; i++ )
            {
                if( aqi[i].hr == S_OK )
                {
                    aqi[i].pItf->Release();
                }
            }
            // Creation failed, so don't allow the
            // client to use the object
            throw XCreateFoStringException();
        }
        m_ctx = ctx;
    }
```

The beginning of the implementation declares a table of the interfaces required to properly construct a faux-object, i.e., IUnknown, IString and IPersist in the case of the FoString. The CoCreateInstanceEx() function uses the CLSID that uniquely identifies the particular implementation of the interfaces required by the client. If the creation succeeds, the table will be filled with interface references, which are cached by the object and used to implement the member function pass-throughs. Otherwise, if the object could not be found or not all of the interfaces are available, interfaces that were successfully acquired are released and an exception is thrown. The exception stops the client from using the object if the implementation of each required interface could not be guaranteed.

In addition to creating a faux-object that creates a COM object, it is useful to be able to create a faux-object given an interface to an existing COM object. Faux-object creation for an existing COM object is supported with a constructor that takes an interface reference and queries for all required interfaces. Here is the implementation of that constructor and the QueryInterfaces() support function is uses:

```
class FoString
{
// Single Interface Constructor
public:
    FoString(IUnknown* punk)
    : m_pIUnknown(0), m_pIString(0), m_pIPersist(0)
    { QueryInterfaces(punk); }

// Helpers
private:
    void QueryInterfaces(IUnknown* punk)
    {
        // Manually query for all required interfaces
        if( !punk ||
```

```
            FAILED(punk->QueryInterface(IID_IUnknown,
                                        (void**)&m_pIUnknown)) ||
            FAILED(punk->QueryInterface(IID_IString,
                                        (void**)&m_pIString)) ||
            FAILED(punk->QueryInterface(IID_IPersist,
                                        (void**)&m_pIPersist)) )
        {
            // If all interfaces aren't available,
            // release those retrieved
            if( m_pIUnknown )
            {
                m_pIUnknown->Release();
            }

            if( m_pIString )
            {
                m_pIString->Release();
            }

            if( m_pIPersist )
            {
                m_pIPersist->Release();
            }

            // Don't allow the client to access
            // an object that isn't properly constructed
            throw XCreateFoStringException();
        }
    }

// Other members removed for clarity
};
```

No matter which constructor is used, the faux-object's destructor releases the cached interfaces via the ReleaseAll() member function when the faux-object goes out of scope:

```
void FoString::ReleaseAll()
{
    if( m_pIUnknown )
    {
        // Pointers are zero'd out to support
        // the assignment operator and the copy
        // constructor described below
        m_pIUnknown->Release(); m_pIUnknown = 0;
        m_pIString->Release(); m_pIString = 0;
```

```
            m_pIPersist->Release(); m_pIPersist = 0;
        }
    }
```

Finally, to stop the programmer from accidentally using COM reference counting on the faux-object, the IUnknown pass-through member functions AddRef() and Release() have been left out. For a client that requires a reference to one of the object's interfaces directly, the faux-object exposes the QueryInterface() member function. The additional lifetime of the underlying COM object represented by this additional outstanding interface reference is the responsibility of the client, using the standard COM lifetime management rules.

Instead of requiring the developer to manage the multiple required interface references, the faux-object manages them. In this way, the object lifetime rules of C++ are successfully mapped onto the reference counting rules of COM. The programmer who requires direct control of the COM object outside of the faux-object is allowed such control without interfering with the interfaces cached by the faux-object.

4.2.3 Provide an implementation guarantee for client-required functionality

The faux-object constructor provides a run-time implementation guarantee for COM objects in a way conceptually similar to the way the C++ compiler provides an implementation guarantee for C++ objects at compile-time. In C++, if the implementation of all of the required member functions is not provided, the compiler error stops the incomplete object from being created.

In COM, if the requested implementation is unavailable or all of the required interfaces are not implemented, the faux-object constructor stops the incomplete object from being used by throwing a C++ exception. When an exception is thrown, the construction of the faux-object is aborted before it is complete, therefore making it impossible for a client to attempt to use an incomplete COM object.

Extending the analogy of C++ compile-time versus COM run-time, a C++ programmer can fix the compile-time error by giving the compiler an object that does implement the missing member functions. Likewise, a COM client is able to catch the exception thrown by a faux-object at run-time and attempt to bind to another implementation. This re-binding process could happen automatically – analogous to a C++ compiler fixing your broken code – or the client could alert the user and request the class identifier for another implementation – analogous to a C++ compiler error message. Either way, the faux-object provides an implementation guarantee that simplifies client code.

4.2.4 Provide C++ type compatibility with individual interfaces

Faux-objects are convenient for a new C++ client using COM objects. However, a great deal of existing code – provided both by the operating system and existing COM clients and COM objects – does not know a thing about faux-objects and depends on references to COM interfaces to provide functionality. To allow a faux-object to be used with these existing routines, some mechanism must be provided to expose the interface references from a faux-object. In other words, to use a faux-object where one of its interface references is expected, the faux-object should be *type compatible* with each of the supported interfaces.

Type compatibility is achieved in C++ via inheritance. A derived class is type compatible with a base class. In the earlier example, a Mule was type compatible with both a Horse and a Donkey because Mule derived from Horse and Donkey. Any function that expected a Horse or a Donkey could be provided with a Mule instead because a Mule is a Horse and a Donkey. The type compatibility relationship is often referred to as an *is-a* relationship. A Mule is-a Horse. A Mule is-a Donkey.

On the other hand, the faux-object clearly implements a *has-a* relationship with each of its cached interfaces. A FoString has-a reference to an IString. A FoString has-a reference to an IPersist interface. A FoString is not an IString interface, but rather uses an IString interface reference to provide its functionality. To simulate the convenience of type compatibility in situations such as these, C++ provides the *typecast operation* [Stroustrup86].

A custom operator allows an object to have an arbitrary meaning for standard operators like + or -. This is handy for allowing normal expressions to be used on user defined types, e.g. adding two instances of a user defined complex number type. A custom typecast operator allows an object to provide an arbitrary meaning for a C++ *cast* operation. A cast is notation that allows the programmer to tell the compiler to ignore the type of the variable and use the programmer-specified type instead. The faux-object will use custom typecast operators to expose each of its supported interfaces. For example, the FoString faux-object exposes three typecast operators:

```
class FoString
{
// Typecast Operators
public:
    operator IUnknown*()
    { return m_pIUnknown; }

    operator IString*()
    { return m_pIString; }

    operator IPersist*()
```

```
    { return m_pIPersist; }

// Other members removed for clarity
};
```

A C++ client using a faux-object can now use an existing COM routine that takes a COM interface pointer by passing in the faux-object. Here's an example of a client with a faux-object calling a function that takes a reference to an IString interface:

```
// Definitions removed for clarity

void ShowString(IString* ps)
{
    char* psz;
    ps->GetText(&psz);

    cout << "Showing: " << psz << endl;

    CoTaskMemFree(psz);
}

void main()
{
    CoInitialize(0);

    try
    {
        FoString    s(CLSID_CoString);
        ShowString(s);
    }
    catch( ... ) {}

    CoUninitialize();
}
```

The reason an operator is used instead of another, more explicit, member function, e.g. GetIString(), or by making the interface reference members public, is for notational convenience. It is easier to write:

```
ShowString(s);
```

then to write:

```
ShowString(s.GetIString());
```

or to write:

```
ShowString(s.m_pIString);
```

Use of the typecast operator also looks more like the standard type compatibility the C++ programmer is used to. However, the disadvantage of the typecast operator is that it does not AddRef() the duplicated interface reference. For input parameters, as shown above, this usage is correct[13]. However, when an interface reference is cached or returned as a result from a function using the typecast operator, the missing AddRef() will cause an inaccuracy in the object's lifetime management. In these scenarios, a faux-object client must perform a manual AddRef() on the resulting interface reference.

It may seem that multiple-inheritance could be considered an attractive alternative to typecast operators when providing type compatibility of faux-objects with the supported interfaces. Indeed, if a faux-object derived from each of the interfaces, it would be type compatible with each of them. In addition, each of the member function implementations would properly be forwarded to the actual COM object when called via the faux-object's base interface. Finally, the additional source code to hand out the interface reference would not be required at all. Unfortunately, adding type compatibility to a faux-object via multiple-inheritance violates the mapping of C++ scoping rules to COM scoping rules.

When a typecast operator is called, a reference to an interface implemented by the actual COM object is provided by the faux-object. Any reference counting can happen by talking directly to the object in the standard COM fashion. However, if multiple-inheritance were used, a reference to the faux-object would be provided to the routine instead of a reference the underlying COM object. Any client caching a reference to a faux-object expecting it to obey the rules of COM scoping would be disappointed when the faux-object went out of C++ scope and left the client with a *dangling pointer*. A dangling pointer is a reference to a C++ object that has gone away. The COM scoping rules avoid this problem using reference counting, but the faux-object obeys the rules of C++ scoping. No number of AddRef()s to a faux-object will keep it around when it has gone out of C++ scope. The typecast operator approach provides type compatibility without leaving the potential for dangling pointers.

---

[13] COM allows AddRef()/Release() pairs to be optimized away when "special knowledge" is available, e.g. when an interface reference is passed as an input parameter.

4.3 Faux-Object Additions

In addition to supporting the four chief requirements, the faux-object implementation that I have developed provides several additional features that enhance the integration with C++.

4.3.1 Copy Constructor & Assignment Operator

A C++ object is allowed to provide its own interpretation of the construction of one object as a copy of another and the assignment of one object over another. The faux-object approach can be extended to provide these services using the appropriate COM interface references.

The copy constructor and the assignment operator are fundamentally linked. A copy constructor is involved when a new C++ object is to be created using the state of an existing C++ object. The assignment operator is involved when an existing C++ object's state is to be replaced by that of another existing C++ object.

The client of a faux-object copy constructor and assignment operator can choose among four broad possibilities:

- **No copy allowed.**
  Disallowing the copy operation can easily be implemented by declaring a private copy constructor and a private assignment operator and by providing no implementations of these member functions.
- **Copy of reference to COM object.**
  The faux-object copy will be another reference to the same underlying COM object. Copy of reference can be implemented by copying the cached interfaces from the source faux-object to the destination.
- **Copy of value of COM object.**
  The faux-object will create another COM object and copy the state of the source COM object to the destination. Copy of value can be implemented using one of the COM serialization interfaces, e.g., IPersistStream or IPersistStorage.
  It should be noted that many COM objects implement serialization via IPersistStorage by caching the IStorage pointer they are given. The current faux-object implementation creates a temporary implementation of IStorage for use during the copying process only. This technique only works properly for objects that take a snapshot of their data from the IStorage pointer and don't cache the interface pointer.
- **Copy of reference to faux-object.**
  Copying of a reference to any C++ object, including a faux-object, involves copying the address of the object only. The copying of an object reference in C++

does not involve the object itself and cannot be customized by the object in any way. More than one reference to any C++ object can lead to the dangling pointer problem and should be handled with care.

4.3.2 Equality Operators

A C++ object is allowed to provide its own interpretation of whether two objects are the same. Two COM objects are defined to be the same if they both provide the same interface pointer value from the result of calling QueryInterface() and asking for IUnknown. Optionally, the faux-object can be implemented to provide an implementation of C++ equality by simply comparing the values of the cached IUnknown interface reference on the left-hand side and the right-hand side of the equality operator. This same technique could be extended to compare a faux-object with a raw COM interface pointer as well.

4.3.3 Minimized Overhead

The faux-object implementation uses several C++ and COM optimization techniques to minimize the time and space overhead:

- **Inline functions.**
  The faux-object member function implementations are all inline. This means that the convenience of the pass-throughs comes at no extra space or function-calling cost.

- **Code generation.**
  Code that is very regular, like the implementation of the faux-object, is often kept in base classes and made table-driven for the sake of flexibility and generalization. The faux-object implementation I have shown "unwinds" most of the table-driven code to its simplest, most efficient form. If this code were hand-written, it would be a tedious process, but since I have built a tool to generate the regular code, it is just as convenient as the base class, table-driven approach.
  In addition, base classes often contain code for every case, even if these features are used not in the derived class. By using code generation, the programmer is allowed to decide what functionality is required at design time without providing extraneous code for cases that are not needed and without coding an implementation by hand.

- **Efficient creation.**
  Because a COM object may be located across a process or a machine boundary, reducing the number of round-trips is an important consideration. The faux-object implementation uses the COM function CoCreateInstanceEx() which can query for all required interfaces in one round trip. CoCreateInstanceEx() is more efficient

than calling CoCreateInstance() for an initial interface and QueryInterface() for all additional interfaces.

- **Efficient usage.**
  For a COM object to be useful, many interfaces are often needed. By providing an implementation guarantee, the faux-object caches those required interfaces. A faux-object changes the typical usage pattern of COM, where an interface is acquired when it is needed, used and then released. For interfaces used more than once, using QueryInterface() every time an interface is needed can mean many more round-trips than are necessary.
  However, for interfaces that are needed infrequently, putting them into a faux-object will mean additional overhead, for example, memory to cache the interface and to hold the proxy-stub pair for remote objects. Such infrequently used interfaces should not be made part of the interface-join of the faux-object, but rather should be acquired as needed via QueryInterface().

4.4 Faux-Object Summary

This chapter has presented an idiom to bridge the gap between object models. The implementation presented was specific to mapping the COM object model onto the C++ object model. The implementation was accomplished using a C++ class that joined multiple interfaces into a single class and used pass-throughs to map member function calls. The faux-object implementation guarantee provides run time verification that all member functions will be properly implemented just like the C++ compiler does. The scoping rules of COM were encapsulated into the faux-object to allow the C++ client the use of regular C++ scoping rules. And to provide a convenience level of compatibility with existing COM-based code, faux-objects provide type compatibility via a C++ typecast operator for every one of the supported interfaces. Finally, the faux-object provides some additional enhancements for convenience and efficiency including various kinds of copy and assignment operators, an equality operation and minimum overhead. And because the faux-object simply uses COM interfaces already exposed from COM implementations, the benefits of faux-objects can be realized on the client-side without any explicit support from the COM object.

Chapter 5.
Faux-Object Generation

It is certainly possible to build objects using the faux-object idiom by hand. However, since the implementation of a faux-object is so regular, it is more convenient to have a tool to build them automatically. I have built such a tool called the FoBuilder.

5.1 FoBuilder

The FoBuilder is a small program that takes input from the user and generates a class using the faux-object idiom. The FoBuilder user interface is shown in Figure 4.

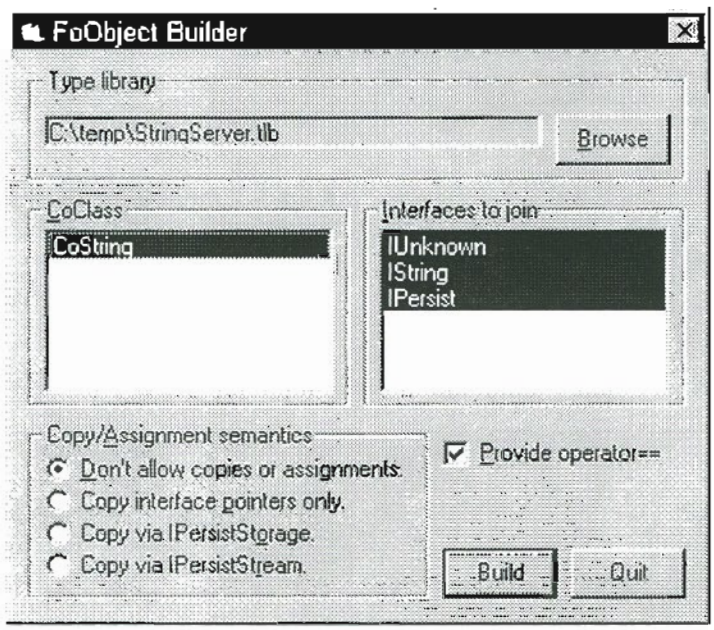


Figure 4: FoBuilder -- a faux-object class generator

The FoBuilder tool allows the user to specify a *Type Library.* A Type Library is a file that describes the classes and interfaces that a *COM object server* supports. A COM object server is a module that provides an implementation of one or more COM classes, i.e. a DLL or an executable. Type Libraries are a standard part of how COM servers describe their supported interfaces and implementations. Type Libraries are defined using the standard COM Interface Definition Language and built using the Microsoft IDL compiler. No special language was created to define interfaces or

implementations for use with the FoBuilder. Rather, the FoBuilder uses Type Libraries already exposed by most COM servers. In this way, COM objects truly do nothing special to support the faux-object idiom. Appendix A shows the IDL used to generate the StringServer.tlb Type Library file.

The pane on the left of the FoBuilder is a list of the classes for the user to choose from. The pane on the right is a list of the interfaces that the selected class supports. From the list of interfaces, the user can choose any combination to join together into a faux-object class. The pane on the bottom left is a list of the possible options for supporting the C++ copy constructor and assignment operator. The Provide operator== option on the right is to provide support in the faux-object for comparing object identity. When the user presses the Build button, a faux-object class is generated and put into a file of the user's choosing. The name of the generated faux-object class is based on the name of the COM class, i.e. selecting CoString will lead to a faux-object class named FoString.

The options indicated in Figure 4 are those used to generate the FoString class used in the previous examples. Appendix B shows the full faux-object class built by the FoBuilder using these options.

5.2 Code Generation

For the FoBuilder to do its job, it needs to know the details of the interfaces it is to generate support for. These details include the member functions of each interface, the type and order of the parameters of each of the member functions and the return code of each of the member functions. This information is all available in the Type Library chosen by the user. The FoBuilder combines the information available in the Type Library and the user-selected options for the copy constructor, assignment operator and equality operator, with a generic faux-object class template to generate a specific faux-object class. This class is then saved in a file of the user's choosing.

5.2.1 Faux-Object Class Template

The current implementation of the FoBuilder generates a faux-object class using a complicated series of if-then-else and select-case statements written in Microsoft Visual Basic. Conceptually, however, the FoBuilder is simply filling in a template based on the information provided by the user. The template is not a C++ class template, however, because C++ templates are not flexible enough to provide the parameterization needed by the FoBuilder. Appendix B shows the conceptual template that the FoBuilder uses to create a faux-object class.

5.2.2 Type Library Deficiencies

A Type Library is generated using the COM Interface Definition Language (IDL). IDL is an unambiguous, text-based description of COM interfaces and implementations. However, because it is text-based, IDL is difficult to parse at runtime. A Type Library is generated from IDL and so is a preprocessed binary file that is guaranteed to be correct. It is therefore much easier to parse a Type Library than an IDL file, which is why the FoBuilder uses Type Libraries.

Unfortunately, for certain interfaces defined in IDL, e.g. IViewObject, not all information is available in a Type Library that is available in the original IDL.[14] Luckily, these interfaces are rare. So, instead of trying to parse IDL, the FoBuilder continues to use Type Libraries, but augments them with knowledge of the standard interfaces for which Type Libraries do not provide sufficient information. When these interfaces are required in a faux-object class, the FoBuilder uses its internal knowledge instead of the information provided in the Type Library to generate the proper support code.

However, the FoBuilder is only augmented to support the standard interfaces that cannot be fully represented in a Type Library. The FoBuilder currently has no support for user-defined interfaces that cannot be fully represented. One way to solve this problem is to extend the FoBuilder to support 3rd party augmentation for user-defined interfaces. This kind of a solution defeats the purpose of using a standard language (IDL) to define interfaces. Another possible solution would be to wait for Microsoft to fix the IDL compiler so that it generates Type Libraries properly. A third solution is to parse IDL. If Microsoft is not forthcoming with a bug fix, a future version of the FoBuilder is likely to be extended to parse IDL.

---

[14] IDL has only recently been expanded to generate Type Libraries. Unfortunately, the Type Library data format is not complete enough to describe interfaces as fully as IDL does.

Chapter 6.
Extended Faux-Object Example

6.1 FoOleObject

While FoStrings are useful for illustration, they are hardly an example of real-world use of the faux-object idiom. To put this idiom to the test, I found an existing COM client application and ported it to use one faux-object class instead of several COM interfaces. The application I found is a sample provided with the OLE Software Development Kit. The SimpCntr sample is an example of a OLE object container that supports in-place activation, menu and toolbar negotiation and serialization. It provides these services using the interfaces exposed by every insertable OLE object: IUnknown, IViewObject, IViewObject2, IPersistStorage and IOleObject. However, instead of caching these interfaces all at once, the SimpCntr sample queries for them as needed. This results in many more round-trips than is optimal and client code that has to check for object functionality again and again. The need for implementation guarantee, interface caching and simplified client code made the faux-object idiom desirable. The version of SimpCntr modified to use faux-objects is called SimpCntr2. Figure 5 shows the results of porting the client from raw COM to using a faux-object class:

Client	Bytes	OLE Object LOC	Total LOC
SimpCntr (no faux-object use)	55,296	50	1848
SimpCntr2 (faux-object use)	56,320	26	1829

Figure 5: Code simplification statistics using a faux-object

Porting the code for one COM object to the faux-object idiom increased the executable size by less than 2%, but reduced the OLE object related lines of code by 48%. Appendix D shows the entire FoOleObject class as generated by the FoBuilder.

6.2 Porting to Faux-Objects

Using the faux-object idiom requires a few simple steps, which are listed below:

1. **Identify the required interfaces.**
   While it is the object's job to provide implementations of COM interfaces, it is the developer's job to decide which of these interfaces are required based on the needs of the client.
   To decide which interfaces were required, I looked through the SimpCntr code. This step is generally easier when you write a COM client yourself, because you know which interfaces you need from an object. Since I didn't design the SimpCntr application, and there were so many COM interfaces referenced and cached, it took longer to decide which interfaces of the embedded OLE object were being used.

2. **Describe the interfaces.**
   In IDL, an interface definition, marked with an `interface` statement, lists the member functions of an interface. Likewise, an implementation definition, marked with a `coclass` statement, lists the interfaces supported by the implementation. The FoBuilder needs a Type Library with a coclass statement that lists all of the required interfaces. Some COM servers provide a Type Library with this information already available. Some COM servers provide an IDL file, which can be used to generate a Type Library. Some COM servers provide neither. In the last case, a minimal IDL file is required on which to run the Microsoft IDL compiler, midl.exe, to generate the required Type Library.
   In the SimpCntr case, neither a Type Library nor an IDL file was provided, so I built my own Type Library using IDL. The following is the complete IDL file needed to build a Type Library with the required interfaces:

```
// All of the interfaces used in SimpCntr were standard
// and defined in the following Microsoft-provided IDL
// files. These definitions are needed to generate
// faux-object pass-throughs functions.
import "unknwn.idl";
import "oleidl.idl";

// The library block defines the information
// to include in a Type Library.
[
    uuid(6BEA8F30-EF00-11cf-939C-86C505000000),
    version(1.0),
    helpstring("SimpCntr2 FoObject Type Library")
]
library SimpCntr2
{
    // This coclass block lists the interfaces
    // supported by this implementation. These
    // interfaces will be listed as possible
    // members of a faux-object interface-join
```

```
    // in the right pane of the FoBuilder.
    [ uuid(6BEA8F31-EF00-11cf-939C-86C505000000) ]
    coclass CoOleObject
    {
        // The definitions of these interfaces are
        // imported into the Type Library from the
        // import statements above.
        [default] interface IUnknown;
        interface IViewObject;
        interface IViewObject2;
        interface IPersistStorage;
        interface IOleObject;
    }
}
```

3. **Build the faux-object class.**
   Once the Type Library is obtained, run the FoBuilder and choose the appropriate options. Again, these options are based on the needs of the client.
   In the case of the faux-object to be generated for the SimpCntr application – FoOleObject – all of the interfaces were selected from the CoOleObject implementation, no copy or assignment was allowed and no equality operator was required.

4. **Use the faux-object class.**
   Instead of using CoCreateInstance(), create instances of the faux-object using the C++ scoping rules. Remember to catch exceptions during creation, as a requested implementation or the required interfaces may not be available.
   In the SimpCntr case, I replaced all of the code that created the OLE object and queried for interfaces with faux-object calls. All of the calls to QueryInterface() went away and all of the rest of the function calls simply worked without change, as the arguments remained the same. Here's an example of the kind of code that I replaced in the SimpCntr example:

```
// Define an interface pointer
LPVIEWOBJECT2 lpViewObject2;
HRESULT hErr;

// Query the OLE object for its implementation of
// IViewObject2 using a cached interface pointer
hErr = m_pSite->m_lpOleObject->
           QueryInterface(IID_IViewObject2,
                          (LPVOID FAR *)&lpViewObject2);

//  If the query succeeded, call a single member
// function of the interface and release it
```

```
if (hErr == NOERROR) {
    lpViewObject2->GetExtent(DVASPECT_CONTENT, -1, NULL,
                             &m_pSite->m_sizel);
    lpViewObject2->Release();
}
```

These six lines of code (not including comments) could be replaced with a single line of code using the `FooleObject` referenced via the `m_pSite` data member `m_lpOleObject`:

```
// Instead of using a cached interface pointer, use
// a faux-object which has already cached all of the
// required interface pointers and provides pass-through
// functions for all of the interfaces' functions.
m_pSite->m_lpOleObject->GetExtent(DVASPECT_CONTENT, -1,
                                  NULL,
                                  &m_pSite->m_sizel);
```

Notice that the actual call to GetExtent() was unchanged. Once the interface management, implementation guarantee and lifetime control code was removed (because It is handled in the faux-object), the client code is greatly simplified.

5. **Test.**
   Normal testing should be used to insure that the assumptions made when generating the faux-object are the proper assumptions. In the SimpCntr case, once the code compiled, it ran flawlessly. All insertable objects that I tested with worked just as they had before.

Chapter 7.
Discussion, Conclusion and Future Work

7.1 Discussion

7.1.1 Faux-Object Cost

One of the additional benefits of the faux-object class was that it provides a bridge between a C++ client and a COM object without adding too much overhead. One of the benefits of the C++ object model is that the memory requirements and the member function call overhead are kept very low compared with other object models. The downside is that there's no "middle layer" in the object model in which to integrate COM. By middle layer, I mean there is no service performed by the language to cover up the differences between the C++ and the COM object models like that provided in Visual Basic. So, to provide a bridge between the C++ and the COM object models, the faux-object adds the middle layer, but what are the costs?

One of the costs is the memory needed for several object references instead of just one, i.e., one reference for every cached interface. Were we able to use multiple inheritance of interfaces, however, this overhead would be same, i.e., one vptr for every base class. Caching multiple references adds no additional overhead.

Another cost is the forwarding of faux-object member function calls to the underlying COM object via interface member functions. However, since we are using the C++ inline function feature, the compiler replaces the faux-object member function call with the call to the interface member function at the caller site. So, faux-object pass-throughs have no additional overhead, although they still have the cost of the indirection required by COM interface member function calls.

A third cost is the additional time it takes to create a COM object and to query for all of the required interfaces. However, since all interfaces can be retrieved in a single round trip, the cost is actually less than the typical COM usage model of querying for an interface every time it is needed.

The caching of COM objects causes one final cost. In the remote case, every interface has the potential to create a proxy-stub pair to perform the communication

required to marshal the member function parameters between the client and the object. For frequently used interfaces, it is more efficient to load the proxy-stub pair and leave it in memory instead of unloading it and reloading it. However, for infrequently used interfaces, this proxy-stub pair consumes unnecessary memory. To minimize this cost, the faux-object should include only the interfaces that are commonly used and required by the client. Those that are optional or not commonly used should be acquired as needed via QueryInterface().

So, the overall cost of the faux-object is small when compared with the code that a client would normally write. The C++ overhead is nearly zero and COM performance can actually improve.

7.1.2 COM vs. CORBA

In many ways the faux-object idiom is similar to the C++ language mapping for CORBA. The CORBA language mapping does a far better job of integration with C++ than raw COM does. It might seem like I could have saved myself a lot of work by just using CORBA.

However, COM has a number of benefits that CORBA does not have:

- COM supports in-process objects with no additional member function overhead. Only a few ORBs support in-process objects.
- COM is supported as a *free* part of every operating system that Microsoft has shipped since Windows 3.1. Windows constitutes approximately 80% of the desktop market and a growing percentage of the server market. In addition, Microsoft and other software vendors are supporting other operating systems, including Apple's MacOS System 7, Sun's Solaris, Hewlett-Packard's HP/UX, Linux and Digital Equipment Corporation's VMS.
- There exist a large number of powerful and inexpensive tools for building COM objects and COM clients. While it has been traditionally true that UNIX development tools are more powerful than Windows development tools, the last few years have turned the tables. Many developers have switched to Windows and tool vendors have concentrated their efforts on this platform more than any other.

In short, by adding the language binding benefits of CORBA to COM, I have made the use of COM easier for the growing number of COM client programmers who wish to program in C++.

7.2 Conclusion

Recall the goal: to bridge the gap between two different object models. Bridging this gap would simplify the client code written using one object model without losing

the functionality of the object written in the object model. The mechanism to bridge the object models is a faux-object – a layer of glue that maps elements of one object model to the other. Specifically, I presented a faux-object implementation scheme to allow C++ clients easy access to COM objects while still minimizing C++ and COM overhead. To make this idiom available without tedious hand coding of each faux-object class, I built the FoBuilder to generate faux-object classes on demand. This thesis has shown two examples of faux-object classes generated using the FoBuilder tool. Each example shows greatly simplified client code and little or no overhead. The faux-object idiom met, and in some cases exceeded, my expectations. I believe that I have solved the basic problem.

7.3 Future Work

The faux-object idiom could be expanded for advanced use by C++ clients to integrate even more of the C++ object model into COM. The following is a list of additional features that could be added:

- **Dead object detection.**
  Because COM objects can live in an address space separate from that of the client, there is the possibility of a communications error between the two. COM provides the ability to catch this class of error by examining the result of calling an interface member function. The faux-object could detect these kinds of errors in the pass-throughs and throw an exception in the event of a communications error. Since this capability would add to the overhead of a faux-object, this functionality should be optional.
- **Static member functions.**
  The C++ object model supports the notion of class member functions as well as object member functions. Class member functions perform operations common to all of the instances of the class, e.g., returning the number of objects. COM provides this facility via a class object that may support interfaces to perform class-level functionality. Given the interfaces of a COM class object, a faux-object could provide this functionality via C++ static member functions.
- **Default arguments.**
  While COM member functions do not support default arguments, C++ member functions do. A faux-object could further simplify client code by supporting default values in the pass-throughs.

- **Support IMultiQI.**
  Modern implementations of COM[15] support an additional interface in the proxy: IMultiQI. IMultiQI allows a client to query for many interfaces at once, much like CoCreateInstanceEx(), and could be used to make construction of a faux-object more efficient when constructing from an interface reference to an existing COM object. However, since most in-process COM objects will not support this interface, the faux-object must support both IMultiQI and IUnknown when constructing a faux-object in this way.

- **More robust copy of value.**
  The current faux-object implementation of copy of value assumes a COM object that does not cache the reference to the IStorage interface passed during serialization. This is not the norm. Most COM objects do cache this reference and are considered "unconstructed" until they are given one. For such objects, the C++ assignment operator has no meaning. The faux-object should be expanded to handle COM objects that support serialization in this manner.

- **Better code generator implementation.**
  Frankly, the implementation of the FoBuilder is not as straightforward as I would like. The FoBuilder can build faux-objects using the idiom as it is currently defined, but expanding it for additional features would be a chore. Adding the features listed above should be a matter of expanding a template and not restructuring the FoBuilder tool itself. I would like a future code generator to be more flexible, more extensible and more robust.

- **Supporting other languages.**
  It would be interesting to take the faux-object idiom and apply it to other object model pairs. For example, the C++ language mapping for CORBA could be cleaned up considerably, I think, using faux-objects. Other interesting combinations would include C++/Java, C++/Perl and Ada/COM. Each of these pairs would have different mapping specifics, but the general technique of building an object in one language whose implementation is bound to an object in other language would remain the same.
  As an example of this possibility, Appendix E shows a faux-object class implementation for C/COM and Appendix F shows a sample client. You may notice that the faux-object usage code is nearly identical to that of C++ except that C does not support exceptions and the faux-object provides global functions instead of member functions.

---

[15] Modern implementations of COM are those that support access to objects across the machine boundary.

Chapter 8.
References

[Almes85] Almes, G., Black, A., Lazowska, E. and Noe, J., "The Eden System: A Technical Review," IEEE Transactions on Software Engineering, vol. SE-11, no. 1, pp. 43-58, January 1985.

[Anderson86] Anderson, D., "Experience with Flamingo: A Distributed, Object-Oriented User Interface System," OOPSLA Proceedings, pp. 177-185, 1986.

[APM91] Architecture Projects Management Ltd, "The ANSA computational model," AR.001, APM Ltd, August 1991.

[Bennett90] Bennett, J., "Experience With Distributed Smalltalk," Software - Practice and Experience, vol. 20, no. 2, pp. 157-180, Feb. 1990.

[Berlin90] Berlin, L., "When Objects Collide: Experiences with Reusing Multiple Class Hierarchies," Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, pp. 181-193, 1990.

[Black86] Black, A., Hutchinson, N., Jul, E. and Levy, H., "Object Structure in the Emerald System," Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, pp. 78-86, 1986.

[Black87] Black, A., Hutchinson, N., Jul E., Levy, H. and Carter, L., "Distribution and Abstract Types in Emerald," IEEE Transactions on Software Engineering, vol. SE-13, no. 1, pp. 65-76, January 1987.

[Bracha90] Bracha, G. and Cook, W., "Mixin-Based Inheritence," Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, pp. 38-45, 1990.

[Brock95] Brockschmidt, K., Inside OLE, 2nd Edition, Microsoft Press, 1995.

[Cardelli85] Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism," ACM Computing Surveys, vol. 17, no. 4, pp. 471-523, December 1985.

[Chen93] Chen, D. and Huang, S., "Interface for reusable software components," Journal of Object-Oriented Programming, pp. 42-53, January, 1993.

[COM95] Microsoft Corp., "The Component Object Model Specification," Microsoft Corp., 1995.

[CPPREF] "Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++," Document No. X3J16/96-0225 WG21/N1043, American National Standards Institute, 1996.

[Curtis97] Curtis, D., "Java, RMI and CORBA," http://www.omg.org/news/wpjava.htm, Object Management Group, 1997.

[Donahue85] Donahue, J. and Demers, A., "Data Types are Values," ACM Transactions on Programming Languages and Systems, vol. 7, no. 3, pp. 426-445, July 1985.

[Flanagan96] Flanagan, D., Java in a Nutshell, O'Reilly & Associates, Inc., 1996.

[Hailpern90] Hailpern, B. and Ossher, H., "Extending Objects to Support Multiple Interfaces and Access Control," IEEE Transactions on Software Engineering, vol. 16, no. 11, pp. 1247-1257, November 1990.

[Helm90] Helm, R., Holland, I. And Gangopadhyay, D., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, pp. 169-180, 1990.

[Java96] Microsoft Corp., Visual J++ Books Online, Microsoft Visual J++ 1.0, Online help pages, Microsoft, 1996.

[Jones86] Jones, M. and Rashid, R., "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, pp. 67-77, 1986.

[Jordan90] Jordan, D., "Implementation Benefits of C++ Language Mechanisms," Communications of the ACM, vol. 33, no. 9, pp. 61-64, 1990.

[Keremitsis95] Keremitsis, E., Fuller, I., "HP Distributed Smalltalk: A Tool For Developing Distributed Applications," Hewlett-Packard Journal, vol. 46, no. 2, pp. 85-92, April 1995.

[Khosh86] Khoshafian, S. and Copeland, G., "Object Identity," OOPSLA Proceedings, pp. 406-416, 1986.

[Korson90] Korson, T. and McGregor, J., "Understanding Object-Oriented: A Unifying Paradigm," Communications of the ACM, vol. 33, no. 9, pp. 40-60, September 1990.

[LaLonde90] LaLonde, W. and Dugh, J., Inside Smalltalk Volume I, Prentice Hall, 1990.

[LaLonde90a] LaLonde, W., Pugh, J., "Preparing To Use The Distributed Facility In IBM Smalltalk," Journal of Object-Oriented Programming, vol. 9, no. 2, pp. 44-48, May 1996.

[Lippman96] Lippman, S., Inside the C++ Object Model, Addison-Wesley, 1996.

[Listkov88] Liskov, B., "Distributed Programming in Argus," Communications of the ACM, vol. 31, no. 3, pp. 300-313, March 1988.

[McGregor90] McGregor, J., "Understanding Object-Oriented: A Unifying Paradigm," Communications of the ACM, vol. 33, no. 9, pp. 40-60, September, 1993.

[RMI96] Sun Microsystems, "Remote Method Invocation Specification," JDK™ 1.1.3 Documentation, http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html, Sun Microsystems, 1996.

[RPC93] Open Software Foundation, AES/Distributed Computing – Remote Procedure Call, Open Software Foundation, 1993.

[Shriv91] Shrivastava, S., Dixon, G. and Parrington, G., "Arjuna: Reliable Distributed System Programming," IEEE Software, pp. 66-73, January 1991.

[Siegel96] Siegel, J., CORBA Fundamentals and Programming, John Wiley & Sons, 1996.

[Snyder86] Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages," Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, pp. 38-45, 1986.

[Stroustrup86] Stroustrup, B., The C++ Programming Language, Addison-Wesley, 1986.

[VB97] Microsoft Corp., Visual Basic Books Online, Microsoft Visual Basic 5.0, Online help pages, Microsoft Corp., 1997.

[WOSA93] Microsoft Corp., "WOSA Backgrounder: Delivering Enterprise Services to the Windows-based Desktop," Microsoft Developer Network, Online help pages, Microsoft Corp., July, 1993.

Appendix A.
StringServer.idl

```
import "unknwn.idl";

[
    uuid(73F86A20-621C-11cf-88D2-00008600A105),
    object
]
interface IString : IUnknown
{
    HRESULT SetText([in, string] const char* szText);
    HRESULT GetText([out, string] char* pszText);
    HRESULT GetLength([out] long* pnLen);
}

[
    uuid(56CA6580-F23F-11cf-88D5-00008600A105),
    version(1.0),
    helpstring("CoString Server")
]
library CoStringLib
{
    [ uuid(0845D620-621A-11cf-88D2-00008600A105) ]
    coclass CoString
    {
        [default] interface IUnknown;
        interface IString;
        interface IPersist;
    }
}
```

Appendix B.
FoString

```
// FoString FoObject class
// (Define necessary interfaces prior to including this file)

#ifndef FOSTRING_H
#define FOSTRING_H

class XCreateFoStringException {};
class XCopyFoStringException {};

class FoString
{

// GUID Constructor
public:
    FoString(REFCLSID clsid, DWORD ctx = CLSCTX_ALL)
    { Create(clsid, ctx); }

// ProgID Constructor
public:
    FoString(LPOLESTR szProgID, DWORD ctx = CLSCTX_ALL)
    {
        CLSID clsid;
        if( SUCCEEDED(CLSIDFromProgID(szProgID, &clsid)) )
        {
            Create(clsid, ctx);
        }
        else
        {
            throw XCreateFoStringException();
        }
    }

// Single Interface Constructor
public:
    FoString(IUnknown* punk)
    : m_pIUnknown(0), m_pIString(0), m_pIPersist(0)
    { QueryInterfaces(punk); }

// Copy Constructor
private:
    FoString(FoString&);

// Assignment Operator
private:
    FoString& operator=(FoString&);
```

```
// Destructor
public:
    virtual ~FoString()
    { ReleaseAll(); }

// Typecast Operators
public:
    operator IUnknown*()
    { return m_pIUnknown; }

    operator IString*()
    { return m_pIString; }

    operator IPersist*()
    { return m_pIPersist; }


// IUnknown Pass-Throughs
public:
    HRESULT QueryInterface(REFIID riid, void** ppv)
    { return m_pIUnknown->QueryInterface(riid, ppv); }

// IString Pass-Throughs
public:
    HRESULT SetText(LPSTR szText)
    { return m_pIString->SetText(szText); }

    HRESULT GetText(LPSTR* pszText)
    { return m_pIString->GetText(pszText); }

    HRESULT GetLength(long* pnLen)
    { return m_pIString->GetLength(pnLen); }


// IPersist Pass-Throughs
public:
    HRESULT GetClassID(struct _GUID* pClassID)
    { return m_pIPersist->GetClassID(pClassID); }


// Helpers
private:
    void Create(REFCLSID clsid, DWORD ctx)
    {
        MULTI_QI aqi[3] = {
                                {&IID_IUnknown, 0, 0},
                                {&IID_IString, 0, 0},
                                {&IID_IPersist, 0, 0},
                            };

        if( CoCreateInstanceEx(clsid, 0, ctx, 0, 3, aqi) == S_OK )
        {
            m_pIUnknown = (IUnknown*)aqi[0].pItf;
            m_pIString = (IString*)aqi[1].pItf;
            m_pIPersist = (IPersist*)aqi[2].pItf;
        }
        else
```

```
        {
            for( int i = 0; i < 3; i++ )
            {
                if( aqi[i].hr == S_OK )
                {
                    aqi[i].pItf->Release();
                }
            }
            throw XCreateFoStringException();
        }
        m_ctx = ctx;
    }

    void QueryInterfaces(IUnknown* punk)
    {
        if( !punk ||
            FAILED(punk->QueryInterface(IID_IUnknown,
                                        (void**)&m_pIUnknown)) ||
            FAILED(punk->QueryInterface(IID_IString,
                                        (void**)&m_pIString)) ||
            FAILED(punk->QueryInterface(IID_IPersist,
                                        (void**)&m_pIPersist)) )
        {
            if( m_pIUnknown )
            {
                m_pIUnknown->Release();
            }

            if( m_pIString )
            {
                m_pIString->Release();
            }

            if( m_pIPersist )
            {
                m_pIPersist->Release();
            }

            throw XCreateFoStringException();
        }
    }

    void ReleaseAll()
    {
        if( m_pIUnknown )
        {
            m_pIUnknown->Release(); m_pIUnknown = 0;
            m_pIString->Release(); m_pIString = 0;
            m_pIPersist->Release(); m_pIPersist = 0;
        }
    }

// Interface Pointer Members
private:
    IUnknown* m_pIUnknown;
    IString* m_pIString;
    IPersist* m_pIPersist;
```

```
// Other cached state
private:
    DWORD m_ctx;
};

#endif // FOSTRING_H
```

Appendix C.
FoBuilder Template

Code in < < > > is expanded by the FoBuilder based on the faux-object class options.

```
// <<faux-object name>> FoObject class
// (Define necessary interfaces prior to including this file)

#ifndef <<faux-object name>>_H
#define <<faux-object name>>_H

class XCreate<<faux-object name>>Exception {};
class XCopy<<faux-object name>>Exception {};

class <<faux-object name>>
{

// GUID Constructor
public:
    <<faux-object name>>(REFCLSID clsid, DWORD ctx = CLSCTX_ALL)
    { Create(clsid, ctx); }

// ProgID Constructor
public:
    <<faux-object name>>(LPOLESTR szProgID, DWORD ctx = CLSCTX_ALL)
    {
        CLSID clsid;
        if( SUCCEEDED(CLSIDFromProgID(szProgID, &clsid)) )
        {
            Create(clsid, ctx);
        }
        else
        {
            throw XCreate<<faux-object name>>Exception();
        }
    }

// Single Interface Constructor
public:
    <<faux-object name>>(IUnknown* punk)
    :
<<for each interface n>>
    m_p<<interface n name>>(0),

    m_pIUnknown(0)
    { QueryInterfaces(punk); }
```

```
// Copy Constructor
<<if copy constructor not allowed>>
private:
    <<faux-object name>>(<<faux-object name>>&);

<<if copy constructor allowed>>
public:
    <<faux-object name>>(<<faux-object name>>& fo)
    {
        Copy(fo);
    }

// Assignment Operator
<<if assignment operator not allowed>>
private:
    <<faux-object name>>& operator=(<<faux-object name>>&);

<<if assignment operator allowed>>
// Assignment Operator
public:
    <<faux-object name>>& operator=(<<faux-object name>>& fo)
    {
        if( m_pIUnknown != fo.m_pIUnknown )
        {
            ReleaseAll();
            Copy(fo);
        }
        return *this;
    }

// Destructor
public:
    virtual ~<<faux-object name>>()
    { ReleaseAll(); }

<<if equality operator allowed>>
// Equality via Object Identifier
public:
    int operator==(const <<faux-object name>>& rhs)
    {
        return (m_pIUnknown == rhs.m_pIUnknown);
    }

// Typecast Operators
public:
    operator IUnknown*()
    { return m_pIUnknown; }

<<for each interface>>
    operator <<interface n name>>*()
    { return m_p<<interface n name>>; }


// IUnknown Pass-Throughs
public:
    HRESULT QueryInterface(REFIID riid, void** ppv)
    { return m_pIUnknown->QueryInterface(riid, ppv); }
```

```
<<for each interface n>>
// <<interface n name>> Pass-Throughs
public:
    <<for each interface n member function>>
    <<member function m result>> <<member function m name>>
        (<<member function m parameters>>)
    { return m_p<<interface n name>>-><<member function m name>>
        (<<member function m parameters>>); }


// Helpers
private:
    void Create(REFCLSID clsid, DWORD ctx)
    {
        MULTI_QI aqi[<<faux-object interfaces>>] = {
                                {&IID_IUnknown, 0, 0},

                            <<for each interface n>>
                                {&IID_<<interface n name>>, 0, 0},
                            };

        if( CoCreateInstanceEx(clsid, 0, ctx, 0,
                                <<faux-object interfaces>>, aqi) == S_OK )
        {
            m_pIUnknown = (IUnknown*)aqi[0].pItf;

        <<for each interface n>>
            m_p<<interface n name>> =
                (<<interface n name>>*)aqi[<<n>>].pItf;
        }
        else
        {
            for( int i = 0; i < <<faux-object interfaces>>; i++ )
            {
                if( aqi[i].hr == S_OK )
                {
                    aqi[i].pItf->Release();
                }
            }
            throw XCreate<<faux-object name>>Exception();
        }
        m_ctx = ctx;
    }

    void QueryInterfaces(IUnknown* punk)
    {
        if( !punk ||

        <<for each interface n>>
            FAILED(punk->QueryInterface(IID_<<interface n name>>,
                      (void**)&m_p<<interface n name>>)) ||

            FAILED(punk->QueryInterface(IID_IUnknown,
                      (void**)&m_pIUnknown)) )
        {
            if( m_pIUnknown )
            {
                m_pIUnknown->Release();
```

```
            }

        <<for each interface n>>
            if( m_p<<interface n name>> )
            {
                m_p<<interface n name>>->Release();
            }

            throw XCreate<<faux-object name>>Exception();
        }
    }

<<if copy/assignment of reference>>
    void Copy(<<faux-object name>>& fo)
    {
        m_pIUnknown = fo.m_pIUnknown;

    <<for each interface n>>
        m_p<<interface n name>> = fo.m_p<<interface n name>>;

        AddRefAll();

        m_ctx = fo.m_ctx;
    }

<<if copy/assignment of value via IPersistStorage>>
    void Copy(<<faux-object name>>& fo)
    {
        if( fo.m_pIUnknown )
        {
            CLSID clsid;
            LPLOCKBYTES pLockBytes = 0;
            LPSTORAGE pStorage = 0;
            const DWORD grfMode = STGM_CREATE | STGM_READWRITE |
                                  STGM_SHARE_EXCLUSIVE;

            try
            {
                if( FAILED(fo.GetClassID(&clsid)) ||
                    FAILED(CreateILockBytesOnHGlobal(0, TRUE,
                                                     &pLockBytes)) ||
                    FAILED(StgCreateDocfileOnILockBytes(pLockBytes,
                                                        grfMode, 0,
                                                        &pStorage)) ||
                    FAILED(fo.Save(pStorage, FALSE)) ||
                    FAILED(fo.SaveCompleted(0)) ||
                    FAILED((Create(clsid, fo.m_ctx), S_OK)) ||
                    FAILED(Load(pStorage)) )
                {
                    throw XCopyFooOleObjectException();
                }
                else
                {
                    pStorage->Release();
                    pLockBytes->Release();
                }
            }
            catch( ... )
```

```
            {
                ReleaseAll();
                if( pStorage ) pStorage->Release();
                if( pLockBytes ) pLockBytes->Release();

                throw;
            }
        }
    }

<<if copy/assignment of value via IPersistStream>>
    void Copy(<<faux-object name>>& fo)
    {
        if( fo.m_pIUnknown )
        {
            CLSID clsid;
            ILockBytes* pLockBytes = 0;
            IStorage* pStorage = 0;
            IStream* pStream = 0;
            const DWORD grfMode = STGM_CREATE | STGM_READWRITE |
                                  STGM_SHARE_EXCLUSIVE;

            try
            {
                if( FAILED(fo.GetClassID(&clsid)) ||
                    FAILED(CreateILockBytesOnHGlobal(0, TRUE,
                                                     &pLockBytes)) ||
                    FAILED(StgCreateDocfileOnILockBytes(pLockBytes,
                                                        grfMode, 0,
                                                        &pStorage)) ||
                    FAILED(pStorage->CreateStream(L"Contents", grfMode, 0,
                                                  0, &pStream)) ||
                    FAILED(fo.Save(pStream, FALSE)) ||
                    FAILED((Create(clsid, fo.m_ctx), S_OK)) ||
                    FAILED(Load(pStream)) )
                {
                    throw XCopyFoStringException();
                }
                else
                {
                    pStream->Release();
                    pStorage->Release();
                    pLockBytes->Release();
                }
            }
            catch( ... )
            {
                ReleaseAll();
                if( pStream ) pStream->Release();
                if( pStorage ) pStorage->Release();
                if( pLockBytes ) pLockBytes->Release();

                throw;
            }
        }
    }

    void ReleaseAll()
```

```
    {
        if( m_pIUnknown )
        {
            m_pIUnknown->Release(); m_pIUnknown = 0;

        <<for each interface n>>
            m_p<<interface n name>>->Release();
            m_p<<interface n name>> = 0;
        }
    }

// Interface Pointer Members
private:
    IUnknown* m_pIUnknown;

<<for each interface n>>
    <<interface n name>>* m_p<<interface n name>>;

// Other cached state
private:
    DWORD m_ctx;
};

#endif // <faux-object name>_H
```

Appendix D.
FoOleObject

```
// FooleObject FoObject class
// (Define necessary interfaces prior to including this file)

#ifndef FOOLEOBJECT_H
#define FOOLEOBJECT_H

class XCreateFooleObjectException {};
class XCopyFooleObjectException {};

class FooleObject
{

// GUID Constructor
public:
    FooleObject(REFCLSID clsid, DWORD ctx = CLSCTX_ALL)
    { Create(clsid, ctx); }

// ProgID Constructor
public:
    FooleObject(LPOLESTR szProgID, DWORD ctx = CLSCTX_ALL)
    {
        CLSID clsid;
        if( SUCCEEDED(CLSIDFromProgID(szProgID, &clsid)) )
        {
            Create(clsid, ctx);
        }
        else
        {
            throw XCreateFooleObjectException();
        }
    }

// Single Interface Constructor
public:
    FooleObject(IUnknown* punk)
    : m_pIUnknown(0), m_pIViewObject(0), m_pIViewObject2(0),
      m_pIPersistStorage(0), m_pIOleObject(0)
    { QueryInterfaces(punk); }

// Copy Constructor
private:
    FooleObject(FooleObject&);

// Assignment Operator
private:
```

```
    FoOleObject& operator=(const FoOleObject&);

// Destructor
public:
    virtual ~FoOleObject()
    { ReleaseAll(); }

// Equality via Object Identifier
public:
    int operator==(const FoOleObject& rhs)
    {
        return (m_pIUnknown == rhs.m_pIUnknown);
    }

// Typecast Operators
public:
    operator IUnknown*()
    { return m_pIUnknown; }

    operator IViewObject*()
    { return m_pIViewObject; }

    operator IViewObject2*()
    { return m_pIViewObject2; }

    operator IPersistStorage*()
    { return m_pIPersistStorage; }

    operator IOleObject*()
    { return m_pIOleObject; }


// IUnknown Pass-Throughs
public:
    HRESULT QueryInterface(REFIID riid, void** ppv)
    { return m_pIUnknown->QueryInterface(riid, ppv); }

// IViewObject Pass-Throughs
public:
    HRESULT Draw(DWORD dwDrawAspect, long lindex, void* pvAspect,
                 struct tagDVTARGETDEVICE* ptd, HDC hdcTargetDev,
                 HDC hdcDraw, struct _RECTL* lprcBounds,
                 struct _RECTL* lprcWBounds,
                 BOOL (STDMETHODCALLTYPE* pfnContinue) (DWORD dwContinue),
                 DWORD dwContinue)
    { return m_pIViewObject->Draw(dwDrawAspect, lindex, pvAspect, ptd,
                                  hdcTargetDev, hdcDraw, lprcBounds,
                                  lprcWBounds, pfnContinue, dwContinue); }

    HRESULT GetColorSet(DWORD dwDrawAspect, long lindex, void* pvAspect,
                        struct tagDVTARGETDEVICE* ptd, void* hicTargetDev,
                        struct tagLOGPALETTE** ppColorSet)
    { return m_pIViewObject->GetColorSet(dwDrawAspect, lindex, pvAspect,
                                         ptd, hicTargetDev, ppColorSet); }

    HRESULT Freeze(DWORD dwDrawAspect, long lindex, void* pvAspect,
                   DWORD* pdwFreeze)
    { return m_pIViewObject->Freeze(dwDrawAspect, lindex, pvAspect,
```

```
                                    pdwFreeze); }

    HRESULT Unfreeze(DWORD dwFreeze)
    { return m_pIViewObject->Unfreeze(dwFreeze); }

    HRESULT SetAdvise(DWORD aspects, DWORD advf,
                      struct IAdviseSink* pAdvSink)
    { return m_pIViewObject->SetAdvise(aspects, advf, pAdvSink); }

    HRESULT GetAdvise(DWORD* pAspects, DWORD* pAdvf,
                      struct IAdviseSink** ppAdvSink)
    { return m_pIViewObject->GetAdvise(pAspects, pAdvf, ppAdvSink); }

// IViewObject2 Pass-Throughs
public:
    HRESULT GetExtent(DWORD dwDrawAspect, long lindex,
                      struct tagDVTARGETDEVICE* ptd, LPSIZEL lpsizel)
    { return m_pIViewObject2->GetExtent(dwDrawAspect, lindex, ptd,
                                        lpsizel); }

// IPersistStorage Pass-Throughs
public:
    HRESULT IsDirty()
    { return m_pIPersistStorage->IsDirty(); }

    HRESULT InitNew(struct IStorage* pstg)
    { return m_pIPersistStorage->InitNew(pstg); }

    HRESULT Load(struct IStorage* pstg)
    { return m_pIPersistStorage->Load(pstg); }

    HRESULT Save(struct IStorage* pStgSave, long fSameAsLoad)
    { return m_pIPersistStorage->Save(pStgSave, fSameAsLoad); }

    HRESULT SaveCompleted(struct IStorage* pStgNew)
    { return m_pIPersistStorage->SaveCompleted(pStgNew); }

    HRESULT HandsOffStorage()
    { return m_pIPersistStorage->HandsOffStorage(); }


// IPersist Pass-Throughs (base of IPersistStorage)
public:
    HRESULT GetClassID(struct _GUID* pClassID)
    { return m_pIPersistStorage->GetClassID(pClassID); }


// IOleObject Pass-Throughs
public:
    HRESULT SetClientSite(struct IOleClientSite* pClientSite)
    { return m_pIOleObject->SetClientSite(pClientSite); }

    HRESULT GetClientSite(struct IOleClientSite** ppClientSite)
    { return m_pIOleObject->GetClientSite(ppClientSite); }
    HRESULT SetHostNames(LPWSTR szContainerApp, LPWSTR szContainerObj)
    { return m_pIOleObject->SetHostNames(szContainerApp, szContainerObj); }

    HRESULT Close(DWORD dwSaveOption)
```

```
{ return m_pIOleObject->Close(dwSaveOption); }

HRESULT SetMoniker(DWORD dwWhichMoniker, struct IMoniker* pmk)
{ return m_pIOleObject->SetMoniker(dwWhichMoniker, pmk); }

HRESULT GetMoniker(DWORD dwAssign, DWORD dwWhichMoniker,
                   struct IMoniker** ppmk)
{ return m_pIOleObject->GetMoniker(dwAssign, dwWhichMoniker, ppmk); }

HRESULT InitFromData(struct IDataObject* pDataObject, long fCreation,
                     DWORD dwReserved)
{ return m_pIOleObject->InitFromData(pDataObject, fCreation,
                                     dwReserved); }

HRESULT GetClipboardData(DWORD dwReserved,
                         struct IDataObject** ppDataObject)
{ return m_pIOleObject->GetClipboardData(dwReserved, ppDataObject); }

HRESULT DoVerb(long iVerb, struct tagMSG* lpmsg,
               struct IOleClientSite* pActiveSite, long lindex,
               HWND hwndParent, struct tagRECT* lprcPosRect)
{ return m_pIOleObject->DoVerb(iVerb, lpmsg, pActiveSite, lindex,
                               hwndParent, lprcPosRect); }

HRESULT EnumVerbs(struct IEnumOLEVERB** ppEnumOleVerb)
{ return m_pIOleObject->EnumVerbs(ppEnumOleVerb); }

HRESULT Update()
{ return m_pIOleObject->Update(); }

HRESULT IsUpToDate()
{ return m_pIOleObject->IsUpToDate(); }

HRESULT GetUserClassID(struct _GUID* pClsid)
{ return m_pIOleObject->GetUserClassID(pClsid); }

HRESULT GetUserType(DWORD dwFormOfType, LPWSTR* pszUserType)
{ return m_pIOleObject->GetUserType(dwFormOfType, pszUserType); }

HRESULT SetExtent(DWORD dwDrawAspect, SIZEL* psizel)
{ return m_pIOleObject->SetExtent(dwDrawAspect, psizel); }

HRESULT GetExtent(DWORD dwDrawAspect, SIZE* psizel)
{ return m_pIOleObject->GetExtent(dwDrawAspect, psizel); }

HRESULT Advise(struct IAdviseSink* pAdvSink, DWORD* pdwConnection)
{ return m_pIOleObject->Advise(pAdvSink, pdwConnection); }

HRESULT Unadvise(DWORD dwConnection)
{ return m_pIOleObject->Unadvise(dwConnection); }

HRESULT EnumAdvise(struct IEnumSTATDATA** ppenumAdvise)
{ return m_pIOleObject->EnumAdvise(ppenumAdvise); }

HRESULT GetMiscStatus(DWORD dwAspect, DWORD* pdwStatus)
{ return m_pIOleObject->GetMiscStatus(dwAspect, pdwStatus); }

HRESULT SetColorScheme(struct tagLOGPALETTE* pLogpal)
```

```
    { return m_pIOleObject->SetColorScheme(pLogpal); }

// Helpers
private:
    void Create(REFCLSID clsid, DWORD ctx)
    {
        MULTI_QI aqi[5] = {
                                {&IID_IUnknown, 0, 0},
                                {&IID_IViewObject, 0, 0},
                                {&IID_IViewObject2, 0, 0},
                                {&IID_IPersistStorage, 0, 0},
                                {&IID_IOleObject, 0, 0},
                            };

        if( CoCreateInstanceEx(clsid, 0, ctx, 0, 5, aqi) == S_OK )
        {
            m_pIUnknown = (IUnknown*)aqi[0].pItf;
            m_pIViewObject = (IViewObject*)aqi[1].pItf;
            m_pIViewObject2 = (IViewObject2*)aqi[2].pItf;
            m_pIPersistStorage = (IPersistStorage*)aqi[3].pItf;
            m_pIOleObject = (IOleObject*)aqi[4].pItf;
        }
        else
        {
            for( int i = 0; i < 5; i++ )
            {
                if( aqi[i].hr == S_OK )
                {
                    aqi[i].pItf->Release();
                }
            }
            throw XCreateFoOleObjectException();
        }
        m_ctx = ctx;
    }

    void QueryInterfaces(IUnknown* punk)
    {
        if( !punk ||
            FAILED(punk->QueryInterface(IID_IUnknown,
                                        (void**)&m_pIUnknown)) ||
            FAILED(punk->QueryInterface(IID_IViewObject,
                                        (void**)&m_pIViewObject)) ||
            FAILED(punk->QueryInterface(IID_IViewObject2,
                                        (void**)&m_pIViewObject2)) ||
            FAILED(punk->QueryInterface(IID_IPersistStorage,
                                        (void**)&m_pIPersistStorage)) ||
            FAILED(punk->QueryInterface(IID_IOleObject,
                                        (void**)&m_pIOleObject)) )
        {
            if( m_pIUnknown )
            {
                m_pIUnknown->Release();
            }

            if( m_pIViewObject )
            {
                m_pIViewObject->Release();
```

```
            }

            if( m_pIViewObject2 )
            {
                m_pIViewObject2->Release();
            }

            if( m_pIPersistStorage )
            {
                m_pIPersistStorage->Release();
            }

            if( m_pIOleObject )
            {
                m_pIOleObject->Release();
            }

            throw XCreateFooOleObjectException();
        }
    }

    void ReleaseAll()
    {
        if( m_pIUnknown )
        {
            m_pIUnknown->Release(); m_pIUnknown = 0;
            m_pIViewObject->Release(); m_pIViewObject = 0;
            m_pIViewObject2->Release(); m_pIViewObject2 = 0;
            m_pIPersistStorage->Release(); m_pIPersistStorage = 0;
            m_pIOleObject->Release(); m_pIOleObject = 0;
        }
    }

// Interface Pointer Members
private:
    IUnknown* m_pIUnknown;
    IViewObject* m_pIViewObject;
    IViewObject2* m_pIViewObject2;
    IPersistStorage* m_pIPersistStorage;
    IOleObject* m_pIOleObject;

// Other cached state
private:
    DWORD m_ctx;
};

#endif // FOOLEOBJECT_H
```

Appendix E.
Faux-Object for C/COM

```
// FoString FoObject "class" for C
// (Define necessary interfaces prior to including this file)

#ifndef FOSTRING_H
#define FOSTRING_H

// FoString structure
typedef struct _FoString
{
    IUnknown*   m_pIUnknown;
    IString*    m_pIString;
    IPersist*   m_pIPersist;
}
FoString;

BOOL FoString_Create(FoString* pThis, REFCLSID clsid, DWORD ctx);
void FoString_Destroy(FoString* pThis);

#define FoString_QueryInterface(pThis, riid, ppv)
IUnknown_QueryInterface((pThis)->m_pIUnknown, riid, ppv)
#define FoString_SetText(pThis, szText) \
          IString_SetText((pThis)->m_pIString, szText)
#define FoString_GetText(pThis, pszText) \
          IString_GetText((pThis)->m_pIString, pszText)
#define FoString_GetLength(pThis, pnLen) \
          IString_GetLength((pThis)->m_pIString, pnLen)
#define FoString_GetClassID(pThis, pClassID) \
       IPersist_GetClassID((pThis)->m_pIPersist, pClassID)

#ifdef FOSTRING_IMPLEMENT

// Constructor
BOOL FoString_Create(FoString* pThis, REFCLSID clsid, DWORD ctx)
{
    MULTI_QI aqi[3] = {
                          {&IID_IUnknown, 0, 0},
                          {&IID_IString, 0, 0},
                          {&IID_IPersist, 0, 0},
                      };

    if( CoCreateInstanceEx(clsid, 0, ctx, 0, 3, aqi) == S_OK )
    {
        pThis->m_pIUnknown = (IUnknown*)aqi[0].pItf;
        pThis->m_pIString = (IString*)aqi[1].pItf;
        pThis->m_pIPersist = (IPersist*)aqi[2].pItf;
    }
```

```
    else
    {
        int i;
        for( i = 0; i < 3; i++ )
        {
            if( aqi[i].hr == S_OK )
            {
                IUnknown_Release(aqi[i].pItf);
            }
        }
        return FALSE;
    }

    return TRUE;
}

// Destructor
void FoString_Destroy(FoString* pThis)
{
    if( pThis->m_pIUnknown )
    {
        IUnknown_Release(pThis->m_pIUnknown); pThis->m_pIUnknown = 0;
        IString_Release(pThis->m_pIString); pThis->m_pIString = 0;
        IPersist_Release(pThis->m_pIPersist); pThis->m_pIPersist = 0;
    }
}

#endif  // FOSTRING_IMPLEMENT
#endif  // FOSTRING_H
```

```
    else
    {
        int i;
        for( i = 0; i < 3; i++ )
        {
            if( aqi[i].hr == S_OK )
            {
                IUnknown_Release(aqi[i].pItf);
            }
        }
        return FALSE;
    }

    return TRUE;
}

// Destructor
void FoString_Destroy(FoString* pThis)
{
    if( pThis->m_pIUnknown )
    {
        IUnknown_Release(pThis->m_pIUnknown); pThis->m_pIUnknown = 0;
        IString_Release(pThis->m_pIString); pThis->m_pIString = 0;
        IPersist_Release(pThis->m_pIPersist); pThis->m_pIPersist = 0;
    }
}

#endif  // FOSTRING_IMPLEMENT
#endif  // FOSTRING_H
```

Appendix F.
Faux-Object for C/COM Client

```
#define _WIN32_DCOM
#define COBJMACROS
#include <windows.h>
#include <stdio.h>
#include <assert.h>
#include "StringServer.h"
#define IID_DEFINED
#include "StringServer_i.c"

#define FOSTRING_IMPLEMENT
#include "FoStringC.h"

void ShowString(IString* ps)
{
    LPSTR   psz = 0;
    IString_GetText(ps, &psz);

    printf("Showing: %s\n", psz);

    if( psz ) CoTaskMemFree(psz);
}

void main()
{
    FoString    s;

    CoInitialize(0);

    if( FoString_Create(&s, &CLSID_CoString, CLSCTX_ALL) )
    {
        LPSTR   psz = 0;
        long    nLen = 0;
        CLSID   clsid;

        // Use IString members in FoString
        FoString_SetText(&s, "Hello, World");
        FoString_GetText(&s, &psz);
        FoString_GetLength(&s, &nLen);

        printf("%s (%d)\n", psz, nLen);

        if( psz ) CoTaskMemFree(psz);

        // Use IPersist members in FoString
        FoString_GetClassID(&s, &clsid);
        assert(IsEqualGUID(&clsid, &CLSID_CoString));
```

```
        // Typecast test
        ShowString(s.m_pIString);

        FoString_Destroy(&s);
    }

    CoUninitialize();
}
```

Bibliographical Sketch

I was born in Duluth, MN on June 2nd in 1969. I obtained my Bachelor's of Computer Science from the University of MN in 1991. I currently specialize in distributed application design and development using Microsoft's Component Object Model. I have been a professional software engineer since 1989, a technical instructor since 1995 and an independent consultant since 1996. I have been awarded a patent in computer telephony integration. I have been published in several magazines and journals and have spoken at several technical conferences. I have co-authored a book entitled "The Downloader's Companion for Windows 95," with Scott Meyers and Catherine Pinch, published in 1995 by Addison-Wesley. I anticipate publication of another book entitled "Windows Telephony Programming," in 1997 also by Addison-Wesley. I can be reached at my email address, csells@sellsbrothers.com, or via my web site, http://www.sellsbrothers.com.