The Component Object Model - COM

Program examples compiled using Visual C++ 6.0 compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this tutorial are listed below. Don't forget to read Tenouk's small disclaimer. The supplementary notes for this tutorial are mfc, COleObjectFactory class and .Net.

Index:

The Component Object Model - COM

The Component Object Model (COM) is the foundation of much of the new Microsoft ActiveX technology, and after five years it's become an integral part of Microsoft Windows. Now we already have COM ++ and most Windows programming will involve COM, so you'd better start learning it now. But where do you begin? You could start with the Microsoft Foundation Class classes for ActiveX Controls, Automation, and OLE, but as useful as those classes are, they obscure the real COM architecture. You've got to start with fundamental theory, and that includes COM and something called an interface.
Here you'll get the theory you need for the next six modules. You'll learn about interfaces and how the MFC library implements interfaces through its macros and interface maps.

ActiveX Technology Background

The terminology is changing as fast as the technology, and not all groups within Microsoft can agree on how to use the terms ActiveX and OLE. Think of ActiveX as something that was created when the "old" **OLE** collided with the **Internet**. ActiveX includes not only those Windows features built on COM (which you'll study in this part of the book) but also the Microsoft Internet Information Server (Internet Information Services is a service used in Windows also having acronym IIS)) family and the WinInet programming interface (covered in Part VI).

Yes, OLE is still here, but once again it stands for Object Linking and Embedding, just as it did in the days of OLE 1. It's just another **subset of ActiveX technology** that includes odds and ends such as drag and drop. Unfortunately (or fortunately, if you have existing code), the MFC source code and the Windows API have not kept current with the naming conventions. As a result, you'll see lots of occurrences of "OLE" and "Ole" in class names and in function names, even though some of those classes and functions go beyond linking and embedding. In this part of the book, you might notice references to the "server" in the code generated by AppWizard. Microsoft has now reserved this term for **database servers** and **Internet servers**; "component" is the new term for OLE servers.
Bookstore computer sections are now full of books on OLE, COM, and ActiveX. We don't claim to offer that level of detail here, but you should come away with a pretty good understanding of COM theory. We've included a closer connection to the **MFC library classes** than you might see in other books, with the exception of MFC Internals (Addison-Wesley, 1996) by George Shepherd and Scot Wingo. The net result should be good preparation for the really heavy-duty ActiveX/COM books, including Kraig Brockschmidt's Inside OLE, 2nd ed. (Microsoft Press, 1995) and Don Box's Essential COM (Addison-Wesley, 1998). A good mid-level book is Dale Rogerson's Inside COM (Microsoft Press, 1997).
One more thing: don't expect this stuff to be easy. Kraig Brockschmidt reported "six months of mental fog" before he started understanding these concepts. A **thorough knowledge of the C++ language** is the minimum prerequisite. Don't be afraid to dig in and write code. Make sure you can do the easy things before getting into advanced areas like multithreaded COM, custom marshaling, and distributed COM (DCOM). As said by Microsoft, COM and .NET are complementary development technologies. See also what Wiki says about COM.

The Component Object Model

COM is an "industry-standard" software architecture supported by Microsoft, Digital Equipment Corporation, and many other companies. It's by no means the only standard. Indeed, it competes directly against other standards, such as The **Common Object Request Broker Architecture**, CORBA from the Open Software Foundation (OSF), maintained by Object Management Group (OMG). Some people are working to establish interoperability between COM and other architectures, but my guess is that COM will become the leading standard.

The Problem That COM Solves

The "problem" is that there's no standard way for Windows program modules to communicate with one another. "But," you say "what about the DLL with its exported functions, Dynamic Data Exchange (DDE), the Windows Clipboard, and the Windows API itself, not to mention legacy standards such as VBX and OLE 1? Aren't they good enough?" Well, no. You can't build an object-oriented operating system for the future out of these **ad hoc**, **unrelated standards**. With the Component Object Model, however, you can, and that's precisely what Microsoft is doing.

The Essence of COM

What's wrong with the old standards? A lot. The Windows API has too large a programming "surface area": **more than 350 separate functions**. VBXs don't work in the 32-bit world. DDE comes with a complicated system of applications, topics, and items. How you call a DLL is totally application-specific. COM provides a unified, expandable, object-oriented communications protocol for Windows that already supports the following features:

- A standard, language-independent way for a Win32 client EXE to load and call a Win32 DLL.
- A general-purpose way for one EXE to control another EXE on the same computer (the DDE replacement).
- A replacement for the VBX control, called an **ActiveX control**.
- A powerful new way for application programs to interact with the operating system.
- Expansion to accommodate new protocols such as Microsoft's OLE DB database interface.
- The distributed COM (DCOM) that allows one EXE to communicate with another EXE residing on a different computer, even if the computers use different microprocessor-chip families, that means different platform or hardware.

So what is COM? That's an easier question to ask than to answer. At DevelopMentor (a training facility for software developers), the party line is that "COM is love." That is, COM is a powerful integrating technology that allows you **to**

mix all sorts of disparate software parts together at runtime. COM allows developers to write software that runs together regardless of issues such as thread-awareness and language choice.
COM is a protocol that connects one software module with another and then drops out of the picture. After the connection is made, the two modules can communicate through a mechanism called an **interface**. Interfaces require no statically or dynamically linked entry points or hard-coded addresses other than the few general-purpose COM functions that start the communication process. An interface (more precisely, a COM interface) is a term that you'll be seeing a lot of.

What Is a COM Interface?

Before digging into the topic of interfaces, let's re-examine the nature of inheritance and polymorphism in normal C++. We'll use a planetary-motion simulation (suitable for NASA or Nintendo) to illustrate C++ inheritance and polymorphism. Imagine a spaceship that travels through our solar system under the influence of the sun's gravity. In ordinary C++, you could declare a `CSpaceship` class and write a constructor that sets the spaceship's initial position and acceleration. Then you could write a non-virtual member function named `Fly()` that implemented Kepler's laws to model the movement of the spaceship from one position to the next, let say, over a period of 0.1 second. You could also write a `Display()` function that painted an image of the spaceship in a window. The most interesting feature of the `CSpaceship` class is that the **interface** of the C++ class (the way the client talks to the class) and the **implementation** are tightly bound. One of the main goals of COM is to separate a class's interface from its implementation.
If we think of this example within the context of COM, the spaceship code could exist as a separate EXE or DLL (the component), which is a COM module. In COM the simulation manager (the client program) can't call `Fly()` or any `CSpaceship` constructor directly: COM provides only a standard global function to gain access to the spaceship object, and then the client and the object use interfaces to talk to one another. Before we tackle real COM, let's build a COM simulation in which both the component and the client code are statically linked in the same EXE file. For our standard global function, we'll invent a function named `GetClassObject()`.
If you want to map this process back to the way MFC works, you can look at `CRuntimeClass`, which serves as a class object for `CObject`-based classes. A class object is a meta-class (either in concept or in form).
In this COM simulation, clients will use this global single abstract function (`GetClassObject()`) for objects of a particular class. In real COM, clients would get a class object first and then ask the class object to manufacture the real object in much the same way MFC does dynamic creation. `GetClassObject()` has the following three parameters:

```
BOOL GetClassObject(int nClsid, int nIid, void** ppvObj);
```

The first `GetClassObject()` parameter, `nClsid`, is a 32-bit integer that uniquely identifies the `CSpaceship` class. The second parameter, `nIid`, is the unique identifier of the interface that we want. The third parameter is a pointer to an interface to the object. Remember that we're going to be dealing with **interfaces** now, (which are different from classes). As it turns out, a class can have several interfaces, so the last two parameters exist to manage interface selection. The function returns `TRUE` if the call is successful.
Now let's go back to the design of `CSpaceship`. We haven't really explained spaceship interfaces yet. A COM interface is a C++ base class (actually, a C++ `struct`) that declares a group of pure virtual functions. These functions completely control some aspect of derived class behavior. For `CSpaceship`, let's write an interface named `IMotion`, which controls the spaceship object's position. For simplicity's sake, we'll declare just two functions, `Fly()` and `GetPosition()`, and we'll keep things uncomplicated by making the position value an integer. The `Fly()` function calculates the position of the spaceship, and the `GetPosition()` function returns a reference to the current position. Here are the declarations:

```
struct IMotion
{
    virtual void Fly() = 0;
    virtual int& GetPosition() = 0;
};

class CSpaceship : public IMotion
{
protected:
```

```
    int m_nPosition;
public:
    CSpaceship() { m_nPosition = 0; }
    void Fly();
    int& GetPosition() { return m_nPosition; }
};
```

The actual code for the spaceship-related functions, including `GetClassObject()` - is located in the component part of the program. The client part calls the `GetClassObject()` function to construct the spaceship and to obtain an `IMotion` pointer. Both parts have access to the `IMotion` declaration at compile time. Here's how the client calls `GetClassObject()`:

```
IMotion* pMot;
GetClassObject(CLSID_CSpaceship, IID_IMotion, (void**) &pMot);
```

Assume for the moment that COM can use the unique integer identifiers `CLSID_CSpaceship` and `IID_IMotion` to construct a spaceship object instead of some other kind of object. If the call is successful, `pMot` points to a `CSpaceship` object that `GetClassObject()` somehow constructs. As you can see, the `CSpaceship` class implements the `Fly()` and `GetPosition()` functions and our main program can call them for the one particular spaceship object, as shown here:

```
int nPos = 50;
pMot->GetPosition() = nPos;
pMot->Fly();
nPos = pMot->GetPosition();
TRACE("new position = %d\n", nPos);
```

Now the spaceship is off and flying. We're controlling it entirely through the `pMot` pointer. Notice that `pMot` is technically not a pointer to a `CSpaceship` object. However, in this case, a `CSpaceship` pointer and an `IMotion` pointer are the same because `CSpaceship` is derived from `IMotion`. You can see how the virtual functions work here: it's classic C++ polymorphism.

Let's make things a little more complex by adding a second interface, `IVisual`, which handles the spaceship's visual representation. One function is enough - `Display()`. Here's the whole base class:

```
struct IVisual
{
    virtual void Display() = 0;
};
```

Are you getting the idea that COM wants you to associate functions in groups? You're not imagining it. But why? Well, in your space simulation, you probably want to include other kinds of objects in addition to spaceships. Imagine that the `IMotion` and `IVisual` interfaces are being used for other classes. Perhaps a `CSun` class has an implementation of `IVisual` but does not have an implementation of `IMotion`, and perhaps a `CSpaceStation` class has other interfaces as well. If you "published" your `IMotion` and `IVisual` interfaces, perhaps other space simulation software companies would adopt them.

Think of an **interface** as **a contract between two software modules**. The idea is that interface declarations never change. If you want to upgrade your spaceship code, you don't change the `IMotion` or the `IVisual` interface; rather, you add a new interface, such as `ICrew`. The existing spaceship clients can continue to run with the old interfaces, and new client programs can use the new `ICrew` interface as well. These client programs can find out at runtime which interfaces a particular spaceship software version supports.

Consider the `GetClassObject()` function as a more powerful alternative to the C++ constructor. With the ordinary constructor, you obtain **one object** with **one batch of member functions**. With the `GetClassObject()` function, you obtain the **object** plus your choice of **interfaces**. As you'll see later, you start with one interface and then use that interface to get other interfaces to the same object.

So how do you program two interfaces for `CSpaceship`? You could use C++ multiple inheritance, but that wouldn't work if two interfaces had the same member function name. The MFC library uses **nested classes instead**, so that's what we'll use to illustrate multiple interfaces on the `CSpaceship` class. Not all C++ programmers are familiar with nested classes, so I'll offer a little help. Here's a first cut at nesting interfaces within the `CSpaceship` class:

```
class CSpaceship
{
protected:
    int m_nPosition;
    int m_nAcceleration;
    int m_nColor;
public:
    CSpaceship()
        { m_nPosition = m_nAcceleration = m_nColor = 0; }
    class XMotion : public IMotion
    {
    public:
        XMotion() { }
        virtual void Fly();
        virtual int& GetPosition();
    } m_xMotion;

    class XVisual : public IVisual
    {
    public:
        XVisual() { }
        virtual void Display();
    } m_xVisual;

    friend class XVisual;
    friend class XMotion;
};
```

It might make sense to make `m_nAcceleration` a data member of `XMotion` and `m_nColor` a data member of `XVisual`. We'll make them data members of `CSpaceship` because that strategy is more compatible with the MFC macros, as you'll see later.
Notice that the implementations of `IMotion` and `IVisual` are contained within the "parent" `CSpaceship` class. In COM, this parent class is known as the **class with object identity**. Be aware that `m_xMotion` and `m_xVisual` are actually embedded data members of `CSpaceship`. Indeed, you could have implemented `CSpaceship` strictly with embedding. Nesting, however, brings to the party two advantages:

1. Nested class member functions can access parent class data members without the need for `CSpaceship` pointer data members, and
2. The nested classes are neatly packaged along with the parent while remaining invisible outside the parent.

Look at the code below for the `GetPosition()` member function.

```
int& CSpaceship::XMotion::GetPosition()
{
    METHOD_PROLOGUE(CSpaceship, Motion) // makes pThis
    return pThis->m_nPosition;
}
```

Notice also the double scope resolution operators, which are necessary for nested class member functions. `METHOD_PROLOGUE` is a one-line MFC macro that uses the C offset of operator to retrieve the offset used in generating a `this` pointer to the parent class, `pThis`. The compiler always knows the offset from the beginning of

parent class data to the beginning of nested class data. `GetPosition()` can thus access the `CSpaceship` data member `m_nPosition`.
Now suppose you have two interface pointers, `pMot` and `pVis`, for a particular `CSpaceship` object. Don't worry yet about how you got these pointers. You can call interface member functions in the following manner:

```
pMot->Fly();
pVis->Display();
```

What's happening under the hood? In C++, each class (at least, each class that has virtual functions and is not an abstract base class) has a **virtual function table**, which is otherwise known as a **vtable**. In this example, that means there are vtables for `CSpaceship::XMotion` and `CSpaceship::XVisual`. For each object, there's a pointer to the object's data, the first element of which is a pointer to the class's vtable. The pointer relationships are shown here.

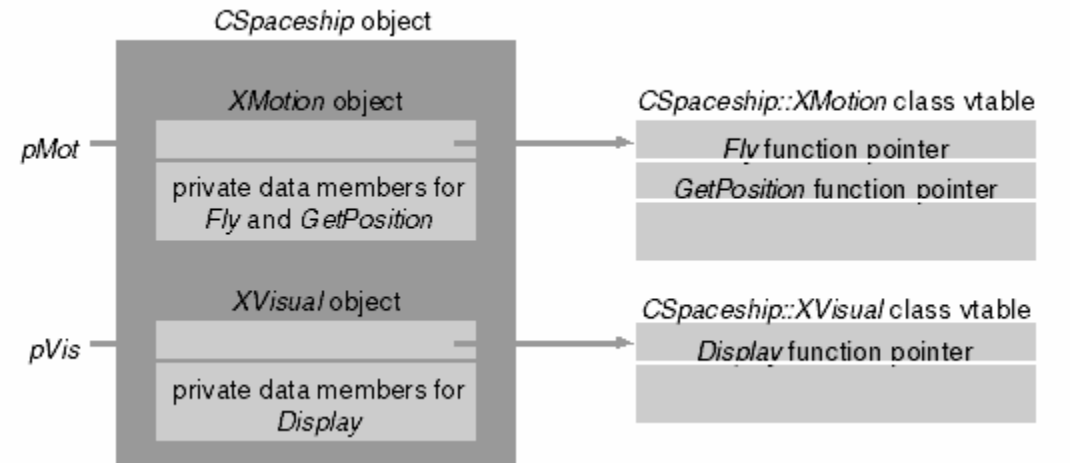

Figure 1: A COM objects pointer relationship.

Theoretically, it's possible to program COM in C. If you look at the Windows header files, you'll see code such as this:

```
#ifdef __cplusplus
        // C++-specific headers
#else
        /* C-specific headers */
#endif
```

In C++, interfaces are declared as C++ **structs**, often with inheritance; in C, they're declared as C **typedef structs** with no inheritance. In C++, the compiler generates vtables for your derived classes; in C, you must "roll your own" vtables, and that gets tedious. It's important to realize, however, that in neither language do the interface declarations have data members, constructors, or destructors. Therefore, you can't rely on the interface having a virtual destructor, but that's not a problem because you never invoke a destructor for an interface.

The `IUnknown` Interface and the `QueryInterface()` Member Function

Let's get back to the problem of how to obtain your interface pointers in the first place. COM declares a special interface named **`IUnknown`** for this purpose. As a matter of fact, all interfaces are derived from `IUnknown`, which has a pure virtual member function, `QueryInterface()`, which returns an interface pointer based on the interface ID you feed it. Once the interface mechanisms are hooked up, the client needs to get an `IUnknown` interface pointer (at the very least) or a pointer to one of the derived interfaces. Here is the new interface hierarchy, with `IUnknown` at the top:

```
struct IUnknown
{
```

```
    virtual BOOL QueryInterface(int nIid, void** ppvObj) = 0;
};

struct IMotion : public IUnknown
{
    virtual void Fly() = 0;
    virtual int& GetPosition() = 0;
};

struct IVisual : public IUnknown
{
    virtual void Display() = 0;
};
```

To satisfy the compiler, we must now add `QueryInterface` implementations in both `CSpaceship::XMotion` and `CSpaceship::XVisual`. What do the vtables look like after this is done? For each derived class, the compiler builds a vtable with the base class function pointers on top, as shown here.

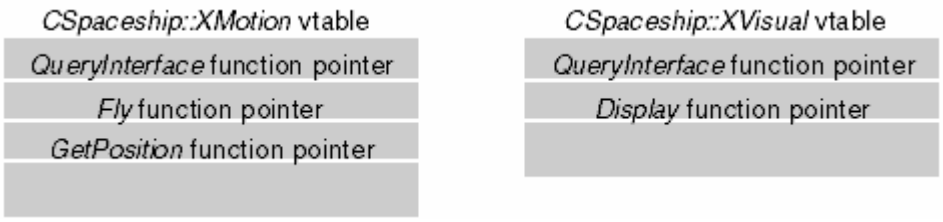

Figure 2: Classes' virtual table (**vtable**).

`GetClassObject()` can get the interface pointer for a given `CSpaceship` object by getting the address of the corresponding embedded object. Here's the code for the `QueryInterface()` function in `XMotion`:

```
BOOL CSpaceship::XMotion::QueryInterface(int nIid, void** ppvObj)
{
    METHOD_PROLOGUE(CSpaceship, Motion)
    switch (nIid) {
    case IID_IUnknown:
    case IID_IMotion:
        *ppvObj = &pThis->m_xMotion;
        break;
    case IID_IVisual:
        *ppvObj = &pThis->m_xVisual;
        break;
    default:
        *ppvObj = NULL;
        return FALSE;
    }
    return TRUE;
}
```

Because `IMotion` is derived from `IUnknown`, an `IMotion` pointer is a valid pointer if the caller asks for an `IUnknown` pointer.
The COM standard demands that `QueryInterface()` return exactly the same `IUnknown` pointer value for `IID_IUnknown`, no matter which interface pointer you start with. Thus, if two `IUnknown` pointers match, you can assume that they refer to the same object. `IUnknown` is sometimes known as the "**`void*`**" of COM because it represents the object's identity.

Below is a `GetClassObject()` function that uses the address of `m_xMotion` to obtain the first interface pointer for the newly constructed CSpaceship object:

```
BOOL GetClassObject(int& nClsid, int& nIid, void** ppvObj)
{
    ASSERT(nClsid == CLSID_CSpaceship);
    CSpaceship* pObj = new CSpaceship();
    IUnknown* pUnk = &pObj->m_xMotion;
    return pUnk->QueryInterface(nIid, ppvObj);
}
```

Now your client program can call `QueryInterface()` to obtain an IVisual pointer, as shown here:

```
IMotion* pMot;
IVisual* pVis;
GetClassObject(CLSID_CSpaceship, IID_IMotion, (void**) &pMot);
pMot->Fly();
pMot->QueryInterface(IID_IVisual, (void**) &pVis);
pVis->Display();
```

Notice that the client uses a `CSpaceship` object, but it never has an actual `CSpaceship` pointer. Thus, the client cannot directly access `CSpaceship` data members even if they're public. Notice also that we haven't tried to delete the spaceship object yet, that will come shortly.
There's a special **graphical representation** for interfaces and COM classes. Interfaces are shown as small circles (or jacks) with lines attached to their class. The `IUnknown` interface, which every COM class supports, is at the top, and the others are on the left. The `CSpaceship` class can be represented something like this.

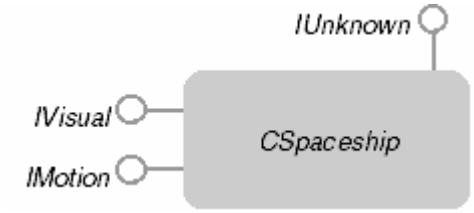

Figure 3: Class and its interfaces.

Reference Counting: The `AddRef()` and `Release()` Functions

COM interfaces don't have virtual destructors, so it isn't cool to write code like the following:

```
delete pMot;  // pMot is an IMotion pointer; don't do this
```

COM has a strict protocol for deleting objects; the two other `IUnknown` virtual functions, `AddRef()` and `Release()`, are the key. Each COM class has a data member, `m_dwRef`, in the MFC library, which keeps track of how many "users" an object has. Each time the component program returns a new interface pointer (as in `QueryInterface()`), the program calls `AddRef()`, which increments `m_dwRef`. When the client program is finished with the pointer, it calls `Release()`. When `m_dwRef` goes to 0, the object destroys itself. Here's an example of a `Release()` function for the `CSpaceship::XMotion` class:

```
DWORD CSpaceship::XMotion::Release()
{
    METHOD_PROLOGUE(CSpaceship, Motion) // makes pThis
    if (pThis->m_dwRef == 0)
        return 0;
    if (--pThis->m_dwRef == 0)
```

```
    {
        delete pThis; // the spaceship object
        return 0;
    }
    return pThis->m_dwRef;
}
```

In MFC COM-based programs, the object's constructor sets `m_dwRef` to 1. This means that it isn't necessary to call `AddRef()` after the object is first constructed. A client program should call `AddRef()`, however, if it makes a copy of an interface pointer.

Class Factories

Object-oriented terminology can get a little fuzzy sometimes. **Smalltalk** programmers, for example, talk about "objects" the way C++ programmers talk about "classes." The COM literature often uses the term "**component object**" to refer to the object plus the code associated with it. COM carries with it the notion of a "**class object**," which is sometimes referred to as a "**class factory**". To be more accurate, it should probably be called an "**object factory**". A COM class object represents the global static area of a specific COM class. Its analogy in MFC is the `CRuntimeClass`. A class object is sometimes called a class factory because it often implements a special COM interface named `IClassFactory`. This interface, like all interfaces, is derived from `IUnknown`. `IClassFactory`'s principal member function is `CreateInstance()`, which in our COM simulation is declared like this:

```
virtual BOOL CreateInstance(int& nIid, void** ppvObj) = 0;
```

Why use a class factory? We've already seen that we can't call the target class constructor directly; we have to let the component module decide how to construct objects. The component provides the class factory for this purpose and thus encapsulates the creation step, as it should. Locating and launching component modules and thus establishing the class factory, is expensive, but constructing objects with `CreateInstance()` is cheap. We can therefore allow a single class factory to create multiple objects.

What does all this mean? It means that we screwed up when we let `GetClassObject()` construct the `CSpaceship` object directly. We were supposed to construct a class factory object first and then call `CreateInstance()` to cause the class factory (object factory) to construct the actual spaceship object.

Let's properly construct the spaceship simulation. First we declare a new class, `CSpaceshipFactory`. To avoid complication, we'll derive the class from `IClassFactory` so that we don't have to deal with nested classes. In addition, we'll add the code that tracks references:

```
struct IClassFactory : public IUnknown
{
    virtual BOOL CreateInstance(int& nIid, void** ppvObj) = 0;
};

class CSpaceshipFactory : public IClassFactory
{
private:
    DWORD m_dwRef;
public:
    CSpaceshipFactory() { m_dwRef = 1; }
    // IUnknown functions
    virtual BOOL QueryInterface(int& nIid, void** ppvObj);
    virtual DWORD AddRef();
    virtual DWORD Release();
    // IClassFactory function
    virtual BOOL CreateInstance(int& nIid, void** ppvObj);
};
```

Next we'll write the `CreateInstance()` member function:

```
BOOL CSpaceshipFactory::CreateInstance(int& nIid, void** ppvObj)
{
    CSpaceship* pObj = new CSpaceship();
    IUnknown* pUnk = &pObj->m_xMotion;
    return pUnk->QueryInterface(nIid, ppvObj);
}
```

Finally, here is the new `GetClassObject()` function, which constructs a class factory object and returns an `IClassFactory` interface pointer.

```
BOOL GetClassObject(int& nClsid, int& nIid, void** ppvObj)
{
    ASSERT(nClsid == CLSID_CSpaceship);
    ASSERT((nIid == IID_IUnknown) || (nIid == IID_IClassFactory));
    CSpaceshipFactory* pObj = new CSpaceshipFactory();
    *ppObj = pObj; // IUnknown* = IClassFactory* = CSpaceship*
}
```

The `CSpaceship` and `CSpaceshipFactory` classes work together and share the same class ID. Now the client code looks like this (without error-checking logic):

```
IMotion* pMot;
IVisual* pVis;
IClassFactory* pFac;
GetClassObject(CLSID_CSpaceship, IID_IClassFactory, (void**) &pFac);
pFac->CreateInstance(IID_IMotion, &pMot);
pMot->QueryInterface(IID_IVisual, (void**) &pVis);
pMot->Fly();
pVis->Display();
```

Notice that the `CSpaceshipFactory` class implements the `AddRef()` and `Release()` functions. It must do this because `AddRef()` and `Release()` are pure virtual functions in the `IUnknown` base class. We'll start using these functions in the next iteration of the program.

The `CCmdTarget` Class

We're still a long way from real MFC COM-based code, but we can take one more step in the COM simulation before we switch to the real thing. As you might guess, some code and data can be "factored out" of our spaceship COM classes into a base class. That's exactly what the MFC library does. The base class is `CCmdTarget`, the standard base class for document and window classes. `CCmdTarget`, in turn, is derived from `CObject`. We'll use `CSimulatedCmdTarget` instead, and we won't put too much in it, only the reference-counting logic and the `m_dwRef` data member. The `CSimulatedCmdTarget` functions `ExternalAddRef()` and `ExternalRelease()` can be called in derived COM classes. Because we're using `CSimulatedCmdTarget`, we'll bring `CSpaceshipFactory` in line with `CSpaceship`, and we'll use a nested class for the `IClassFactory` interface.

We can also do some factoring out inside our `CSpaceship` class. The `QueryInterface()` function can be "delegated" from the nested classes to the outer class helper function `ExternalQueryInterface()`, which calls `ExternalAddRef()`. Thus, each `QueryInterface()` function increments the reference count, but `CreateInstance()` calls `ExternalQueryInterface()`, followed by a call to `ExternalRelease()`. When the first interface pointer is returned by `CreateInstance()`, the spaceship object has a reference count of 1. A subsequent `QueryInterface()` call increments the count to 2, and in this case, the client must call `Release()` twice to destroy the spaceship object.

One last thing, we'll make the class factory object a global object. That way we won't have to call its constructor. When the client calls `Release()`, there isn't a problem because the class factory's reference count is 2 by the time the client

receives it. The `CSpaceshipFactory` constructor sets the reference count to 1, and `ExternalQueryInterface()`, called by `GetClassObject()`, sets the count to 2.

The MYMFC28A Example: A Simulated COM

Listings 1, 2, 3, and 4 show code for a working "simulated COM" program, MYMFC28A. This is a **Win32 Console Application** (without the MFC library) that uses a class factory to construct an object of class `CSpaceship`, calls its interface functions, and then releases the spaceship. The **Interface.h** header file, shown in Listing 1, contains the `CSimulatedCmdTarget` base class and the interface declarations that are used by both the client and component programs. The **Spaceship.h** header file shown in Listing 2 contains the spaceship-specific class declarations that are used in the component program. **Spaceship.cpp**, shown in Listing 3, is the component that implements `GetClassObject()`; **Client.cpp**, shown in Listing 4, is the client that calls `GetClassObject()`. What's phony here is that both **client** and **component code** are linked within the same **mymfc28A.exe** program. Thus, our simulated COM is not required to make the connection at runtime. You'll see how that's done later in this module.

The MYMFC28A From Scratch

The following are the steps to build MYMFC28A project.

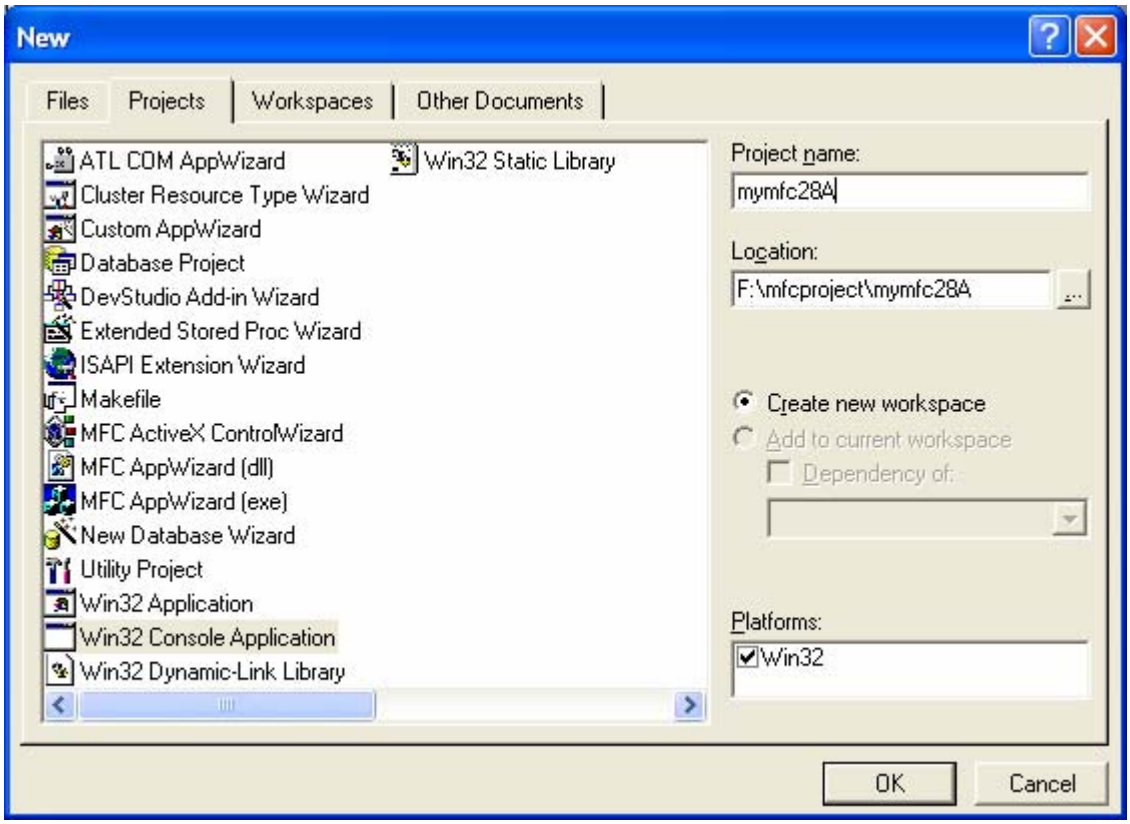

Figure 4: MYMFC28A COM project-AppWizard new project.

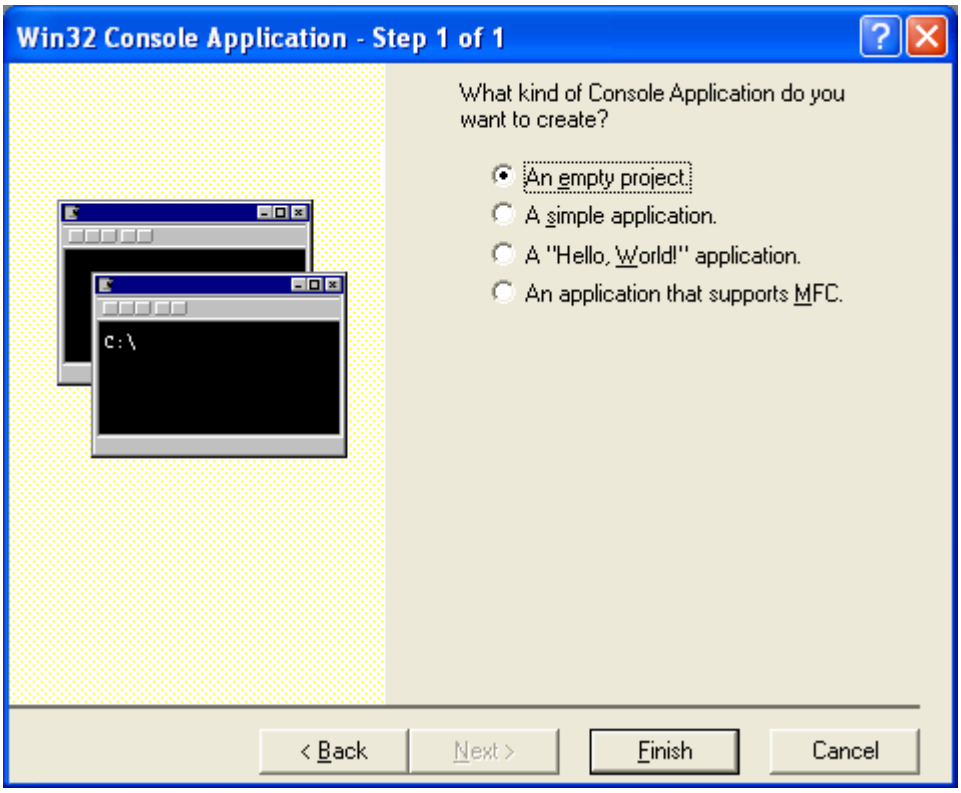

Figure 5: MYMFC28A COM project-AppWizard step 1 of 1.

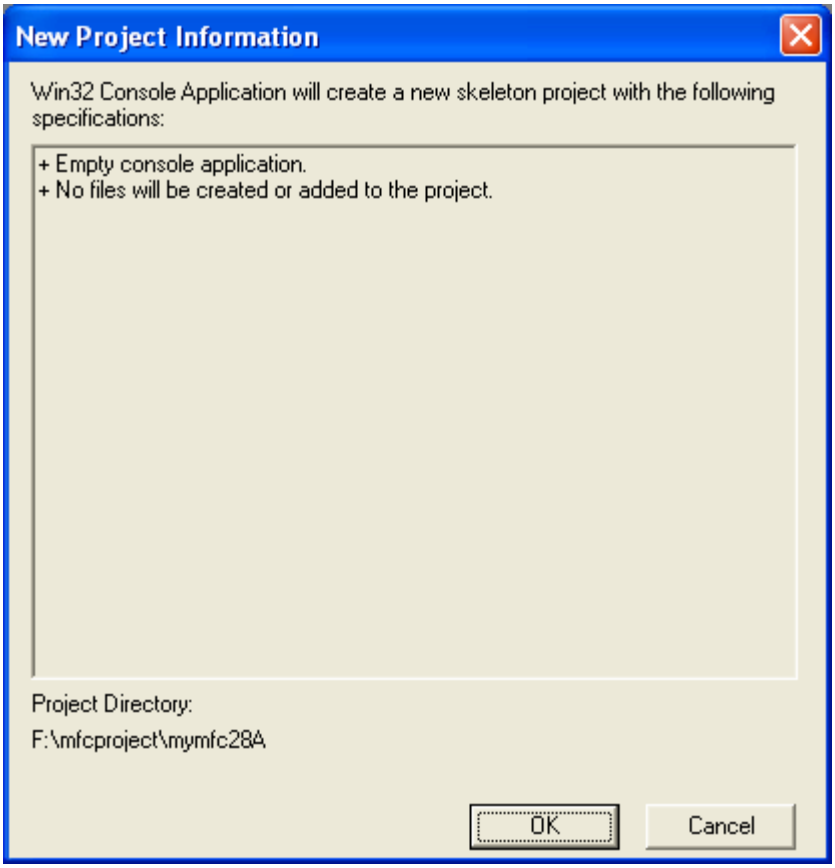

Figure 6: MYMFC28A COM project summary.

Now we are going to add new header and source files to the project.

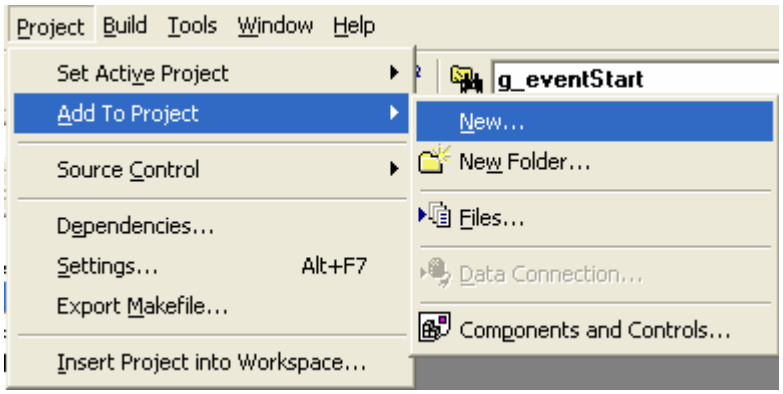

Figure 7: Adding new files to project.

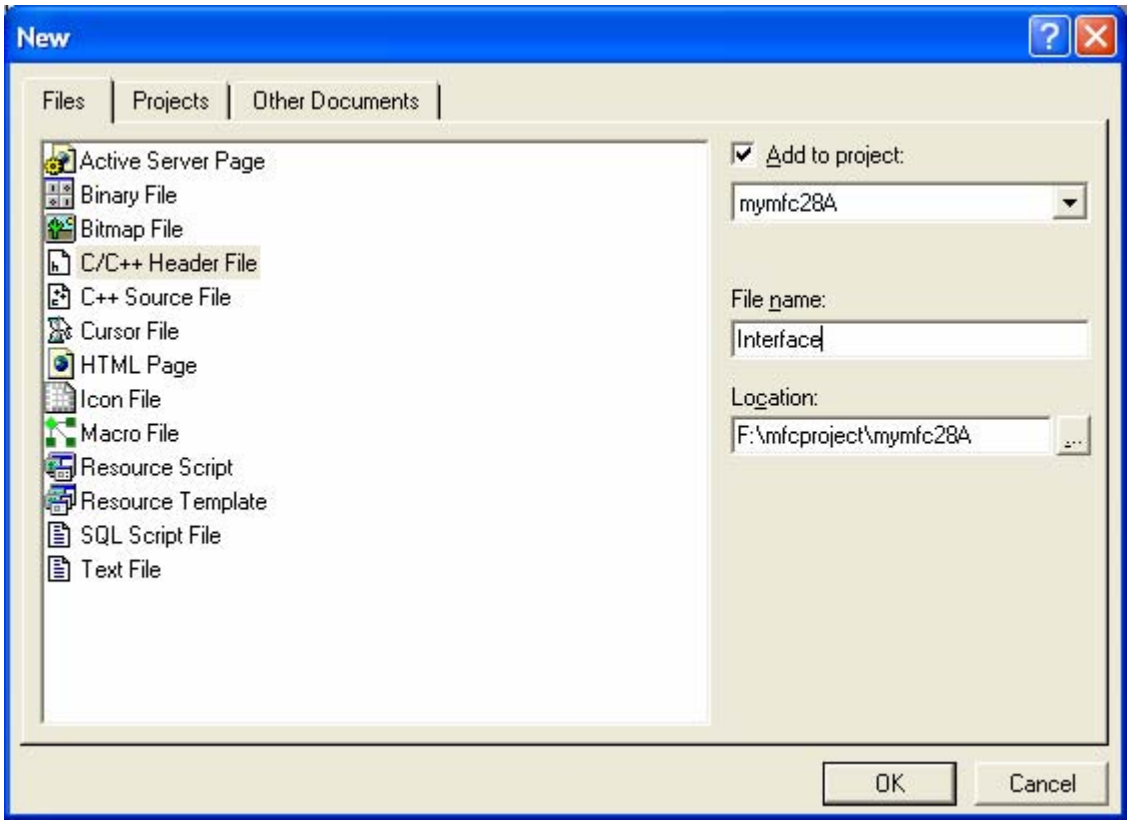

Figure 8: Adding new header file to project.

Then copy and paste the **Interface.h** code given in the listing. Repeat the same step for other files: **Spaceship.h**, **Spaceship.cpp** and **Client.cpp**.

Build and run the program.

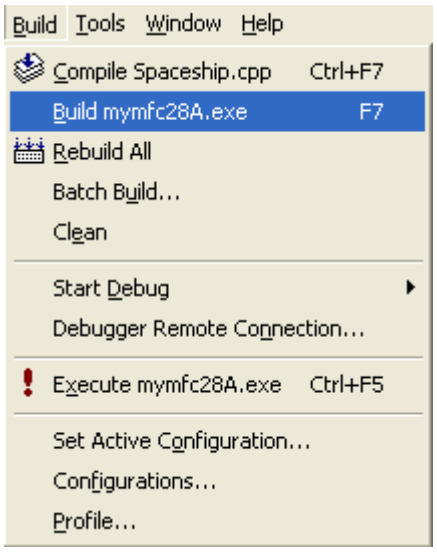

Figure 9: Building Visual C++ project.

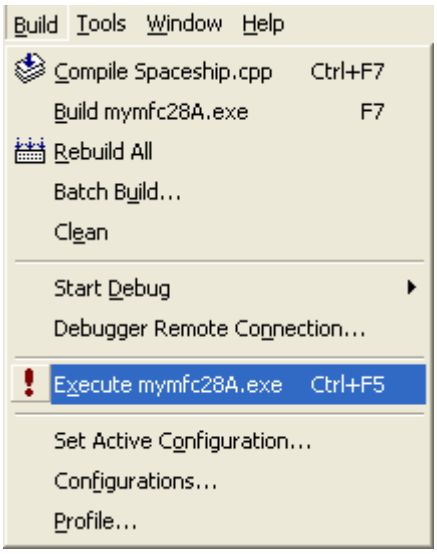

Figure 10: Running Visual C++ project.

The following is the MYMFC28A output screen, depicting the program activities.

```
Entering CSimulatedCmdTarget ctor at 00427930
Entering IUnknown ctor at 00427938
Entering IClassFactory ctor at 00427938
Entering XClassFactory ctor at 00427938
Entering CSpaceshipFactory ctor at 00427930

Entering client main()

Entering CSpaceshipFactory::ExternalQueryInterface()--nIid =  1

Entering CSpaceshipFactory::XClassFactory::CreateInstance()
Entering CSimulatedCmdTarget ctor at 00430060
Entering IUnknown ctor at 00430074
Entering IMotion ctor at 00430074
```

```
Entering XMotion ctor at 00430074
Entering IUnknown ctor at 00430078
Entering IVisual ctor at 00430078
Entering XVisual at ctor
Entering CSpaceship ctor at 00430060

Entering CSpaceship::ExternalQueryInterface()--nIid = 0
Entering CSimulatedCmdTarget::ExternalRelease()--RefCount = 2

Entering CSpaceship::XMotion::QueryInterface()--nIid =  2

Entering CSpaceship::ExternalQueryInterface()--nIid = 2

Entering CSpaceship::XMotion::QueryInterface()--nIid =  3

Entering CSpaceship::ExternalQueryInterface()--nIid = 3

main(): pUnk = 00430074, pMot = 00430074, pDis = 00430078


Entering CSpaceship::XMotion::Fly()
this pointer = 00430074, pThis pointer = 00430060
m_nPosition = 100
m_nAcceleration = 101

Entering CSpaceship::XMotion::GetPosition()
this pointer = 00430074, pThis pointer = 00430060
m_nPosition = 100
m_nAcceleration = 101
nPos = 100

Entering CSpaceship::XVisual::Display()
this pointer = 00430078, pThis pointer = 00430060
m_nPosition = 100
m_nColor = 102

Entering CSpaceshipFactory::XClassFactory::Release()
Entering CSimulatedCmdTarget::ExternalRelease()--RefCount = 2

Entering CSpaceship::XMotion::Release()
Entering CSimulatedCmdTarget::ExternalRelease()--RefCount = 3

Entering CSpaceship::XMotion::Release()
Entering CSimulatedCmdTarget::ExternalRelease()--RefCount = 2

Entering CSpaceship::XVisual::Release()
Entering CSimulatedCmdTarget::ExternalRelease()--RefCount = 1
deleting
Entering CSpaceship dtor at 00430060
Entering CSimulatedCmdTarget dtor at 00430060
Entering CSpaceshipFactory dtor at 00427930
Entering CSimulatedCmdTarget dtor at 00427930
Press any key to continue
```

INTERFACE.H

```
// definitions that make our code look like MFC code
#define BOOL   int
#define DWORD  unsigned int
#define TRUE   1
#define FALSE  0
#define TRACE  printf
```

```
#define ASSERT assert

//----------definitions and macros---------------------------------
#define CLSID_CSpaceship    10
#define IID_IUnknown        0
#define IID_IClassFactory   1
#define IID_IMotion         2
#define IID_IVisual         3

// this macro for 16-bit Windows only
#define METHOD_PROLOGUE(theClass, localClass) \
    theClass* pThis = ((theClass*)((char*)(this) - \
    offsetof(theClass, m_x##localClass))); \


BOOL GetClassObject(int nClsid, int nIid, void** ppvObj);

//----------interface declarations---------------------------------
struct IUnknown
{
    IUnknown() { TRACE("Entering IUnknown ctor at %p\n", this); }
    virtual BOOL QueryInterface(int nIid, void** ppvObj) = 0;
    virtual DWORD Release() = 0;
    virtual DWORD AddRef() = 0;
};

struct IClassFactory : public IUnknown
{
    IClassFactory()
        { TRACE("Entering IClassFactory ctor at %p\n", this); }
    virtual BOOL CreateInstance(int nIid, void** ppvObj) = 0;
};

struct IMotion : public IUnknown
{
    IMotion() { TRACE("Entering IMotion ctor at %p\n", this); }
    virtual void Fly() = 0; // pure
    virtual int& GetPosition() = 0;
};
struct IVisual : public IUnknown
{
    IVisual() { TRACE("Entering IVisual ctor at %p\n", this); }
    virtual void Display() = 0;
};

class CSimulatedCmdTarget // 'simulated' CSimulatedCmdTarget
{
public:
    DWORD m_dwRef;

protected:
    CSimulatedCmdTarget()
       {
        TRACE("Entering CSimulatedCmdTarget ctor at %p\n", this);
        m_dwRef = 1; // implied first AddRef
    }
    virtual ~CSimulatedCmdTarget()
        { TRACE("Entering CSimulatedCmdTarget dtor at %p\n", this); }
    DWORD ExternalRelease()
       {
    TRACE("Entering CSimulatedCmdTarget::ExternalRelease()--RefCount = %ld\n",
m_dwRef);
        if (m_dwRef == 0)
```

```
            return 0;
        if(--m_dwRef == 0L)
              {
            TRACE("deleting\n");
            delete this;
            return 0;
        }
        return m_dwRef;
    }
    DWORD ExternalAddRef() { return ++m_dwRef; }
};
```

Listing 1: The **Interface.h** file.

SPACESHIP.H

```
class CSpaceship;

//----------class declarations---------------------------------------
class CSpaceshipFactory : public CSimulatedCmdTarget
{
public:
    CSpaceshipFactory()
        { TRACE("Entering CSpaceshipFactory ctor at %p\n", this); }
    ~CSpaceshipFactory()
        { TRACE("Entering CSpaceshipFactory dtor at %p\n", this); }
    BOOL ExternalQueryInterface(int lRid, void** ppvObj);
    class XClassFactory : public IClassFactory
    {
    public:
        XClassFactory()
            { TRACE("Entering XClassFactory ctor at %p\n", this); }
        virtual BOOL QueryInterface(int lRid, void** ppvObj);
        virtual DWORD Release();
        virtual DWORD AddRef();
        virtual BOOL CreateInstance(int lRid, void** ppvObj);
    } m_xClassFactory;
    friend class XClassFactory;
};
class CSpaceship : public CSimulatedCmdTarget
{
private:
    int m_nPosition; // We can access these from
                     //  all the interfaces
    int m_nAcceleration;
    int m_nColor;
public:
    CSpaceship() {
        TRACE("Entering CSpaceship ctor at %p\n", this);
        m_nPosition = 100;
        m_nAcceleration = 101;
        m_nColor = 102;
    }
    ~CSpaceship()
        { TRACE("Entering CSpaceship dtor at %p\n", this); }
    BOOL ExternalQueryInterface(int lRid, void** ppvObj);
    class XMotion : public IMotion
    {
    public:
        XMotion()
            { TRACE("Entering XMotion ctor at %p\n", this); }
```

```
        virtual BOOL QueryInterface(int lRid, void** ppvObj);
        virtual DWORD Release();
        virtual DWORD AddRef();
        virtual void Fly();
        virtual int& GetPosition();
    } m_xMotion;

class XVisual : public IVisual
    {
    public:
        XVisual() { TRACE("Entering XVisual at ctor\n"); }
        virtual BOOL QueryInterface(int lRid, void** ppvObj);
        virtual DWORD Release();
        virtual DWORD AddRef();
        virtual void Display();
    } m_xVisual;

    friend class XVisual;  // These must be at the bottom!
    friend class XMotion;
    friend class CSpaceshipFactory::XClassFactory;
};
```

Listing 2: The **Spaceship.h** file.

SPACESHIP.CPP

```
#include <stdio.h>
#include <stddef.h> // for offsetof in METHOD_PROLOGUE
#include <ASSERT.h>
#include "Interface.h"
#include "Spaceship.h"

CSpaceshipFactory g_factory;

//----------member functions----------------------------------------
BOOL CSpaceshipFactory::ExternalQueryInterface(int nIid, void** ppvObj)
{
    TRACE("\nEntering CSpaceshipFactory::ExternalQueryInterface()--nIid =  %d\n",
nIid);
    switch (nIid)
       {
              case IID_IUnknown:
              case IID_IClassFactory:
                     *ppvObj = &m_xClassFactory;
                     break;
              default:
                     *ppvObj = NULL;
        return FALSE;
    }
    ExternalAddRef();
    return TRUE;
}

BOOL CSpaceshipFactory::XClassFactory::QueryInterface(int nIid, void** ppvObj)
{
    TRACE("\nEntering CSpaceshipFactory::XClassFactory::QueryInterface()--nIid =
%d\n", nIid);
    METHOD_PROLOGUE(CSpaceshipFactory, ClassFactory) // makes pThis
    return pThis->ExternalQueryInterface(nIid, ppvObj); // delegate to
                                          //  CSpaceshipFactory
}
```

```
BOOL CSpaceshipFactory::XClassFactory::CreateInstance(int nIid, void** ppvObj)
{
    TRACE("\nEntering CSpaceshipFactory::XClassFactory::CreateInstance()\n");
    METHOD_PROLOGUE(CSpaceshipFactory, ClassFactory) // makes pThis
    CSpaceship* pObj = new CSpaceship();
    if (pObj->ExternalQueryInterface(nIid, ppvObj))
       {
        pObj->ExternalRelease(); // balance reference count
        return TRUE;
    }
    return FALSE;
}

DWORD CSpaceshipFactory::XClassFactory::Release()
{
    TRACE("\nEntering CSpaceshipFactory::XClassFactory::Release()\n");
    METHOD_PROLOGUE(CSpaceshipFactory, ClassFactory) // makes pThis
    return pThis->ExternalRelease(); // delegate to CSimulatedCmdTarget
}

DWORD CSpaceshipFactory::XClassFactory::AddRef()
{
    TRACE("\nEntering CSpaceshipFactory::XClassFactory::AddRef()\n");
    METHOD_PROLOGUE(CSpaceshipFactory, ClassFactory) // makes pThis
    return pThis->ExternalAddRef(); // delegate to CSimulatedCmdTarget
}

BOOL CSpaceship::ExternalQueryInterface(int nIid, void** ppvObj)
{
    TRACE("\nEntering CSpaceship::ExternalQueryInterface()--nIid = %d\n", nIid);
    switch (nIid)
       {
              case IID_IUnknown:
              case IID_IMotion:
                     *ppvObj = &m_xMotion; // Both IMotion and IVisual are derived
                     break;                // from IUnknown, so either pointer will
do
              case IID_IVisual:
                     *ppvObj = &m_xVisual;
                     break;
              default:
                     *ppvObj = NULL;
        return FALSE;
    }
    ExternalAddRef();
    return TRUE;
}

BOOL CSpaceship::XMotion::QueryInterface(int nIid, void** ppvObj)
{
    TRACE("\nEntering CSpaceship::XMotion::QueryInterface()--nIid =  %d\n", nIid);
    METHOD_PROLOGUE(CSpaceship, Motion) // makes pThis
    return pThis->ExternalQueryInterface(nIid, ppvObj); // delegate to
                                                        //  CSpaceship
}

DWORD CSpaceship::XMotion::Release()
{
    TRACE("\nEntering CSpaceship::XMotion::Release()\n");
    METHOD_PROLOGUE(CSpaceship, Motion) // makes pThis
    return pThis->ExternalRelease(); // delegate to CSimulatedCmdTarget
}
```

```
DWORD CSpaceship::XMotion::AddRef()
{
    TRACE("\nEntering CSpaceship::XMotion::AddRef()\n");
    METHOD_PROLOGUE(CSpaceship, Motion) // makes pThis
    return pThis->ExternalAddRef(); // delegate to CSimulatedCmdTarget
}

void CSpaceship::XMotion::Fly()
{
    TRACE("\nEntering CSpaceship::XMotion::Fly()\n");
    METHOD_PROLOGUE(CSpaceship, Motion) // makes pThis
    TRACE("this pointer = %p, pThis pointer = %p\n", this, pThis);
    TRACE("m_nPosition = %d\n", pThis->m_nPosition);
    TRACE("m_nAcceleration = %d\n", pThis->m_nAcceleration);
}

int& CSpaceship::XMotion::GetPosition()
{
    TRACE("\nEntering CSpaceship::XMotion::GetPosition()\n");
    METHOD_PROLOGUE(CSpaceship, Motion) // makes pThis
    TRACE("this pointer = %p, pThis pointer = %p\n", this, pThis);
    TRACE("m_nPosition = %d\n", pThis->m_nPosition);
    TRACE("m_nAcceleration = %d\n", pThis->m_nAcceleration);
    return pThis->m_nPosition;
}

BOOL CSpaceship::XVisual::QueryInterface(int nIid, void** ppvObj)
{
    TRACE("\nEntering CSpaceship::XVisual::QueryInterface()--nIid =  \
          %d\n", nIid);
    METHOD_PROLOGUE(CSpaceship, Visual) // makes pThis
    return pThis->ExternalQueryInterface(nIid, ppvObj); // delegate to
                                                        //  CSpaceship
}

DWORD CSpaceship::XVisual::Release()
{
    TRACE("\nEntering CSpaceship::XVisual::Release()\n");
    METHOD_PROLOGUE(CSpaceship, Visual) // makes pThis
    return pThis->ExternalRelease(); // delegate to CSimulatedCmdTarget
}

DWORD CSpaceship::XVisual::AddRef()
{
    TRACE("\nEntering CSpaceship::XVisual::AddRef()\n");
    METHOD_PROLOGUE(CSpaceship, Visual) // makes pThis
    return pThis->ExternalAddRef(); // delegate to CSimulatedCmdTarget
}

void CSpaceship::XVisual::Display()
{
    TRACE("\nEntering CSpaceship::XVisual::Display()\n");
    METHOD_PROLOGUE(CSpaceship, Visual) // makes pThis
    TRACE("this pointer = %p, pThis pointer = %p\n", this, pThis);
    TRACE("m_nPosition = %d\n", pThis->m_nPosition);
    TRACE("m_nColor = %d\n", pThis->m_nColor);
}

//----------simulates COM component ----------------------------------
// In real COM, this would be DllGetClassObject(), which would be called
//  whenever a client called CoGetClassObject()
```

```
BOOL GetClassObject(int nClsid, int nIid, void** ppvObj)
{
    ASSERT(nClsid == CLSID_CSpaceship);
    ASSERT((nIid == IID_IUnknown) || (nIid == IID_IClassFactory));
    return g_factory.ExternalQueryInterface(nIid, ppvObj);
    // Refcount is 2, which prevents accidental deletion
}
```

Listing 3: The **Spaceship.cpp** file.

```
CLIENT.CPP

#include <stdio.h>
#include <stddef.h> // for offsetof in METHOD_PROLOGUE
#include <assert.h>
#include "interface.h"


//----------main program-----------------------------------------------
int main() // simulates OLE client program
{
    TRACE("\nEntering client main()\n");
    IUnknown* pUnk; // If you declare these void*, you lose type-safety
    IMotion* pMot;
    IVisual* pVis;
    IClassFactory* pClf;

    GetClassObject(CLSID_CSpaceship, IID_IClassFactory, (void**) &pClf);

    pClf->CreateInstance(IID_IUnknown, (void**) &pUnk);
    pUnk->QueryInterface(IID_IMotion, (void**) &pMot); // All three
    pMot->QueryInterface(IID_IVisual, (void**) &pVis); //  pointers
                                                       //  should work

    TRACE("\nmain(): pUnk = %p, pMot = %p, pDis = %p\n\n", pUnk, pMot, pVis);

    // Test all the interface virtual functions
    pMot->Fly();
    int nPos = pMot->GetPosition();
    TRACE("nPos = %d\n", nPos);
    pVis->Display();

    pClf->Release();
    pUnk->Release();
    pMot->Release();
    pVis->Release();
    return 0;
}
```

Listing 4: The **Client.cpp** file.

Real COM with the MFC Library

So much for simulations. Now we'll get ready to convert the spaceship example to genuine COM. You need to acquire a little more knowledge before we start, though. First you must learn about the `CoGetClassObject()` function, then you must learn how COM uses the **Windows Registry to load the component**, and then you have to understand the difference between an **in-process component** (a DLL) and an **out-of-process component** (an EXE or a DLL running as a surrogate). Finally, you must become familiar with the MFC macros that support nested classes.

The net result will be an **MFC regular DLL component** that contains all the `CSpaceship` code with the `IMotion` and `IVisual` **interfaces**. A regular MFC library Windows application acts as the **client**. It loads and runs the component when the user selects a menu item.

The COM `CoGetClassObject()` Function

In our simulation, we used a phony function named `GetClassObject()`. In real COM, we use the global `CoGetClassObject()` function. **`Co`** stands for "component object". Compare the following prototype to the `GetClassObject()` function you've seen already:

```
STDAPI CoGetClassObject(REFCLSID rclsid, DWORD dwClsContext,
    COSERVERINFO* pServerInfo, REFIID riid, LPVOID* ppvObj)
```

The interface pointer goes in the `ppvObj` parameter, and `pServerInfo` is a pointer to a machine on which the class object is instantiated (`NULL` if the machine is local). The types `REFCLSID` and `REFIID` are references to 128-bit `GUID`s (globally unique identifiers for COM classes and interfaces). `STDAPI` indicates that the function returns a 32-bit value of type `HRESULT`.
The standard `GUID`s (for example, those `GUID`s naming interfaces that Microsoft has already created) are defined in the Windows libraries that are dynamically linked to your program. `GUID`s for custom classes and interfaces, such as those for spaceship objects, must be defined in your program in this way:

```
// {692D03A4-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IMotion = {0x692d03a4, 0xc689, 0x11ce, {0xb3, 0x37,
0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e}};
```

If the `dwClsContext` parameter is `CLSCTX_INPROC_SERVER`, the COM subsystem looks for a DLL. If the parameter is `CLSCTX_LOCAL_SERVER`, COM looks for an EXE. The two codes can be combined to select either a DLL or an EXE, selected in order of performance. For example, **inproc** servers are fastest because everybody shares the same address space. Communication EXE servers are considerably slower because the interprocess calls involve data copying as well as many thread context switches. The return value is an `HRESULT` value, which is 0 (`NOERROR`) if no error occurs. Another COM function, `CoCreateInstance()`, combines the functionality of `CoGetClassObject()` and `IClassFactory::CreateInstance`.

COM and the Windows Registry

In the MYMFC28A example, the component was **statically linked** to the client, a clearly bogus circumstance. In real COM, the component is either a DLL or a separate EXE. When the client calls the `CoGetClassObject()` function, COM steps in and finds the correct component, which is located somewhere on disk. How does COM make the connection? It looks up the class's **unique 128-bit class ID number** in the Windows **Registry**. Thus, the class must be registered permanently on your computer.
If you run the Windows **Regedit** program (**Regedt32** in Microsoft Windows NT 4/5), you'll see a screen similar to the one shown in Figure 11. This figure shows subfolders for four class IDs, three of which are class IDs associated with DLLs (**InprocServer32**) and one of which is a class ID associated with an EXE (**LocalServer32**). The `CoGetClassObject()` function looks up the class ID in the Registry and then loads the DLL or EXE as required.
What if you don't want to track those ugly class ID numbers in your client program? No problem. COM supports another type of registration database entry that translates a human-readable program ID into the corresponding class ID. Figure 12 shows the Registry entries. The COM function `CLSIDFromProgID()` reads the database and performs the translation.

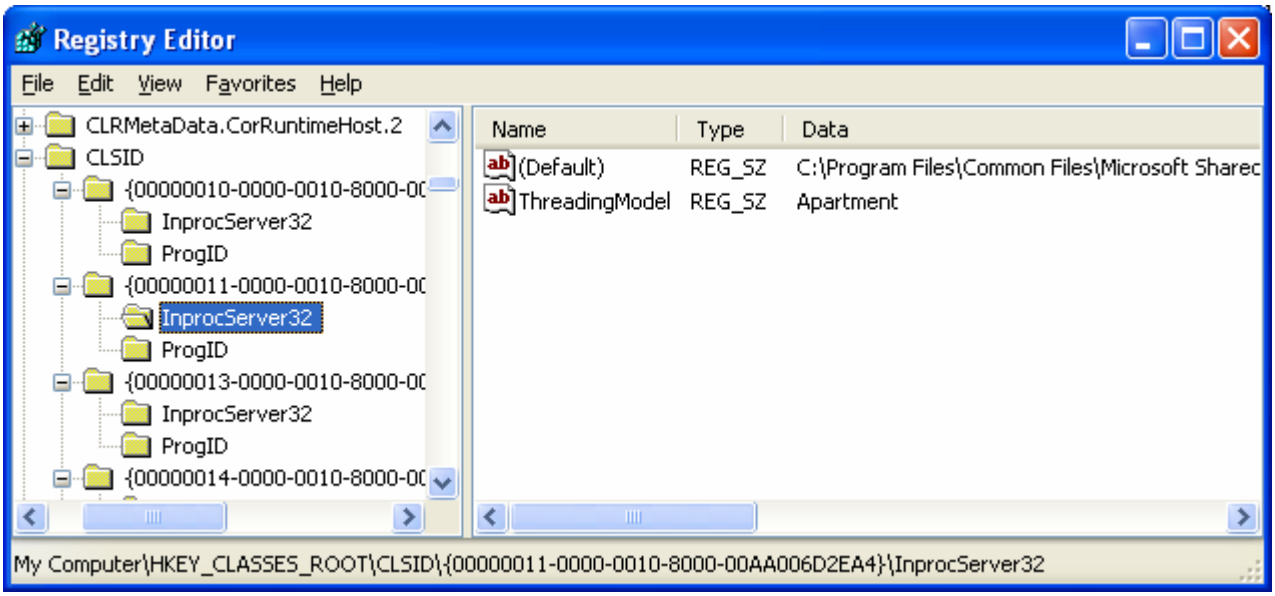

Figure 11: Subfolders of four class IDs in the Registry.

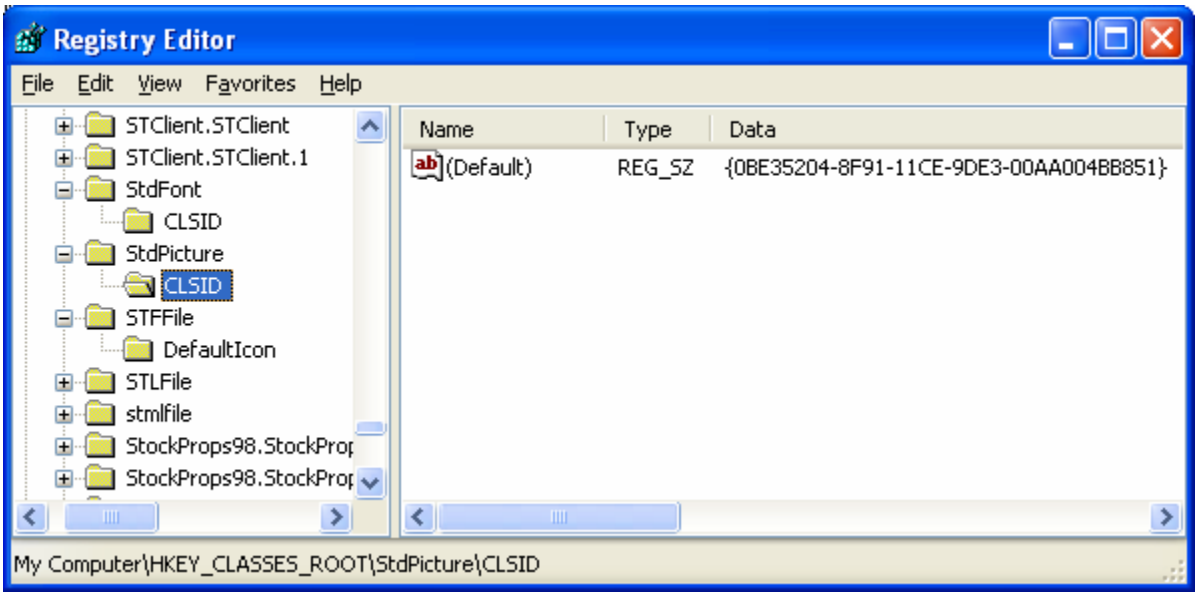

Figure 12: Human-readable program IDs in the Registry.

The first `CLSIDFromProgID()` parameter is a string that holds the program ID, but it's not an ordinary string. This is your first exposure to **double-byte characters** in COM. All string parameters of COM functions (except Data Access Objects (DAOs)) are **Unicode** character string pointers of type `OLECHAR*`. Your life is going to be made miserable because of the constant need to convert between double-byte strings and ordinary strings. If you need a double-byte literal string, prefix the string with an `L` character, like this:

```
CLSIDFromProgID(L"Spaceship", &clsid);
```

How does the registration information get into the Registry? You can program your component application to call Windows functions that directly update the Registry. The MFC library conveniently wraps these functions with the function `COleObjectFactory::UpdateRegistryAll`, which finds all your program's global class factory objects and registers their names and class IDs.

Runtime Object Registration

You've just seen how the Windows Registry registers COM classes on disk. Class factory objects also must be registered. It's unfortunate that the word "register" is used in both contexts. Objects in out-of-process component modules are registered at runtime with a call to the COM `CoRegisterClassObject()` function and the registration information is maintained in memory by the Windows DLLs. If the factory is registered in a mode that permits a single instance of the component module to create multiple COM objects, COM can use an existing process when a client calls `CoGetClassObject()`.

How a COM Client Calls an In-Process Component

We're beginning with a DLL component instead of an EXE component because the program interactions are simpler. I'll show pseudocode here because you're going to be using the MFC library classes, which hide much of the detail.

Client

```
CLSID clsid;
IClassFactory* pClf;
IUnknown* pUnk;
CoInitialize(NULL);  // Initialize COM
CLSIDFromProgID("component_name", &clsid);
```

COM

COM uses the Registry to look up the class ID from "`component_name`".

Client

```
CoGetClassObject(clsid, CLSCTX_INPROC_SERVER, NULL, IID_IClassFactory, (void**)
&pClf);
```

COM

COM uses the class ID to look for a component in memory.

```
if (component DLL is not loaded already)
{
     COM gets DLL filename from the Registry
     COM loads the component DLL into process memory
}
```

DLL Component

```
if (component just loaded)
{
     Global factory objects are constructed
     DLL's InitInstance called (MFC only)
}
```

COM

COM calls DLL's global exported `DllGetClassObject()` with the `CLSID` value that was passed to `CoGetClassObject()`.

DLL Component

`DllGetClassObject()` returns `IClassFactory*`.

COM

COM returns `IClassFactory*` to client.

Client

```
pClf->CreateInstance (NULL, IID_IUnknown, (void**) &pUnk);
```

DLL Component

Class factory's `CreateInstance()` function called (called directly, through component's **vtable**).
Constructs object of "componentname" class.
Returns requested interface pointer.

Client

```
pClf->Release();
pUnk->Release();
```

DLL Component

"`component_name`" `Release()` is called through vtable.

```
     if (refcount == 0)
     {
        Object destroys itself
     }
```

Client

```
CoFreeUnusedLibraries();
```

COM

COM calls DLL's global exported `DllCanUnloadNow()`.

DLL Component

`DllCanUnloadNow()` called if (all DLL's objects destroyed)

```
{
     return TRUE
}
```

Client

```
CoUninitialize();  // COM frees the DLL if DllCanUnloadNow returns TRUE just
                   // prior to exit
```

COM

COM releases resources.

Client

Client exits.

DLL Component

Windows unloads the DLL if it is still loaded and no other programs are using it.

Some important points to note: first, the DLL's exported `DllGetClassObject()` function is called in response to the client's `CoGetClassObject()` call. Second, the class factory interface address returned is the actual physical address of the class factory vtable pointer in the DLL. Third, when the client calls `CreateInstance()`, or any other interface function, the call is direct (through the component's vtable).
The COM linkage between a client EXE and a component DLL is quite efficient, as efficient as the linkage to any C++ virtual function in the same process, plus the full C++ parameter and return type-checking at compile time. The only penalty for using ordinary DLL linkage is the extra step of looking up the class ID in the Registry when the DLL is first loaded.

How a COM Client Calls an Out-of-Process Component

The COM linkage to a separate EXE component is more complicated than the linkage to a DLL component. The EXE component is in a different process, or possibly on a different computer. Don't worry, though. Write your programs as if a direct connection existed. COM takes care of the details through its remoting architecture, which usually involves Remote Procedure Calls (RPCs).
In an RPC, the client makes calls to a special DLL called a proxy. The proxy sends a stream of data to a stub, which is inside a DLL in the component's process. When the client calls a component function, the proxy alerts the stub by sending a message to the component program, which is processed by a hidden window. The mechanism of converting parameters to and from data streams is called **marshaling**.
If you use standard interfaces (those interfaces defined by Microsoft) such as `IClassFactory` and `IPersist` (an interface we haven't seen yet but will appear when we examine COM persistence), the proxy and stub code, which implements marshaling, is provided by the Windows `OLE32` DLL. If you invent your own interfaces, such as `IMotion` and `IVisual`, you **need to write the proxies and stubs yourself**. Fortunately, creating proxy and stub classes only involves defining your interfaces in **Interface Definition Language** (IDL) and compiling the code produced by the **Microsoft Interface Definition Language** (MIDL) compiler.
Here's the pseudocode interaction between an EXE client and an EXE component. Compare it to the DLL version found above. Notice that the client-side calls are exactly the same.

Client

```
CLSID clsid;
IClassFactory* pClf;
IUnknown* pUnk;
CoInitialize(NULL);  // Initialize COM
CLSIDFromProgID("component_name", &clsid);
```

COM

COM uses the Registry to look up the class ID from "`component_name`"

Client

```
CoGetClassObject(clsid, CLSCTX_LOCAL_SERVER, NULL, IID_IClassFactory, (void**)
&pClf);
```

COM

COM uses the class ID to look for a component in memory.

```
    if (component EXE is not loaded already, or
    if we need another instance)
```

```
{
     COM gets EXE filename from the Registry
     COM loads the component EXE
}
```

EXE Component

```
if (just loaded)
{
     Global factory objects are constructed
     InitInstance called (MFC only)
     CoInitialize(NULL);
     for each factory object
     {
          CoRegisterClassObject(...);
          Returns IClassFactory* to COM
     }
     }
```

COM

COM returns the requested interface pointer to the client (client's pointer is not the same as the component's interface pointer).

Client

```
pClf->CreateInstance(NULL, IID_IUnknown, (void**) &pUnk);
```

EXE Component

Class factory's `CreateInstance()` function called (called indirectly through marshaling).
Constructs object of "componentname" class.
Returns requested interface pointer indirectly.

Client

```
pClf->Release();
pUnk->Release();
```

EXE Component

"`component_name`" `Release()` is called indirectly.

```
if (refcount == 0) {
     Object destroys itself
}
if (all objects released) {
     Component exits gracefully
}
```

Client

```
CoUninitialize();  // just prior to exit
```

COM

COM calls `Release()` for any objects this client has failed to release.

EXE Component

Component exits.

COM

COM releases resources.

Client

Client exits.

As you can see, COM plays an important role in the communication between the **client** and the **component**. COM keeps an in-memory list of class factories that are in active EXE components, but it does not keep track of individual COM objects such as the `CSpaceship` object. Individual COM objects are responsible for updating the reference count and for destroying themselves through the `AddRef()`/`Release()` mechanism. COM does step in when a client exits. If that client is using an out-of-process component, COM "listens in" on the communication and keeps track of the reference count on each object. COM disconnects from component objects when the client exits. Under certain circumstances, this causes those objects to be released. Don't depend on this behavior, however. Be sure that your client program releases all its interface pointers prior to exiting.

The MFC Interface Macros

In MYMFC28A, you saw nested classes used for interface implementation. The MFC library has a set of macros that automate this process. For the `CSpaceship` class, derived from the real MFC `CCmdTarget` class, you use the macros shown here inside the declaration.

```
BEGIN_INTERFACE_PART(Motion, IMotion)
    STDMETHOD_(void, Fly) ();
    STDMETHOD_(int&, GetPosition) ();
END_INTERFACE_PART(Motion)

BEGIN_INTERFACE_PART(Visual, IVisual)
    STDMETHOD_(void, Display) ();
END_INTERFACE_PART(Visual)

DECLARE_INTERFACE_MAP()
```

The `INTERFACE_PART` macros generate the nested classes, adding `X` to the first parameter to form the class name and adding `m_x` to form the embedded object name. The macros generate prototypes for the specified interface functions plus prototypes for `QueryInterface()`, `AddRef()`, and `Release()`.
The `DECLARE_INTERFACE_MAP` macro generates the declarations for a table that holds the IDs of all the class's interfaces. The `CCmdTarget::ExternalQueryInterface` function uses the table to retrieve the interface pointers. In the `CSpaceship` implementation file, use the following macros:

```
BEGIN_INTERFACE_MAP(CSpaceship, CCmdTarget)
    INTERFACE_PART(CSpaceship, IID_IMotion, Motion)
    INTERFACE_PART(CSpaceship, IID_IVisual, Visual)
END_INTERFACE_MAP()
```

These macros build the interface table used by `CCmdTarget::ExternalQueryInterface`. A typical interface member function looks like this:

```
STDMETHODIMP_(void) CSpaceship::XMotion::Fly()
{
```

```
    METHOD_PROLOGUE(CSpaceship, Motion)
    pThis->m_nPosition += 10;
    return;
}
```

Don't forget that you must implement all the functions for each interface, including `QueryInterface()`, `AddRef()`, and `Release()`. Those three functions can delegate to functions in `CCmdTarget`. The `STDMETHOD_` and `STDMETHODIMP_` macros declare and implement functions with the `__stdcall()` parameter passing convention, as required by COM. These macros allow you to specify the return value as the first parameter. Two other macros, `STDMETHOD` and `STDMETHODIMP`, assume an `HRESULT` return value.

The MFC `COleObjectFactory` Class

In the simulated COM example, you saw a `CSpaceshipFactory` class that was hard-coded to generate `CSpaceship` objects. The MFC library applies its dynamic creation technology to the problem. Thus, a single class, aptly named `COleObjectFactory`, can create objects of any class specified at runtime. All you need to do is use macros like these in the class declaration:

```
DECLARE_DYNCREATE(CSpaceship)
DECLARE_OLECREATE(CSpaceship)
```

And use macros like these in the implementation file:

```
IMPLEMENT_DYNCREATE(CSpaceship, CCmdTarget)
// {692D03A3-C689-11CE-B337-88EA36DE9E4E}
IMPLEMENT_OLECREATE(CSpaceship, "Spaceship", 0x692d03a3, 0xc689, 0x11ce,
    0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e)
```

The `DYNCREATE` macros set up the standard dynamic creation mechanism. The `OLECREATE` macros declare and define a global object of class `COleObjectFactory` with the specified unique `CLSID`. In a DLL component, the exported `DllGetClassObject()` function finds the specified class factory object and returns a pointer to it based on global variables set by the `OLECREATE` macros. In an EXE component, initialization code calls the static `COleObjectFactory::RegisterAll`, which finds all factory objects and registers each one by calling `CoRegisterClassObject()`. The `RegisterAll()` function is called also when a DLL is initialized. In that case, it merely sets a flag in the factory object(s).
We've really just scratched the surface of MFC's COM support. If you need more details, be sure to refer to Shepherd and Wingo's MFC Internals (Addison-Wesley, 1996) or MSDN online documentation.

AppWizard/ClassWizard Support for COM In-Process Components

AppWizard isn't optimized for creating COM DLL components, but you can fool it by requesting a regular DLL with **Automation** support. The following functions in the project's main source file are of interest:

```
BOOL CMymfc28BApp::InitInstance()
{
    COleObjectFactory::RegisterAll();
    return TRUE;
}

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllGetClassObject(rclsid, riid, ppv);
}
```

```
STDAPI DllCanUnloadNow(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModule_State());
    return AfxDllCanUnloadNow();
}

STDAPI DllRegisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    COleObjectFactory::UpdateRegistryAll();
    return S_OK;
}
```

The three global functions are exported in the project's **DEF** file. By calling MFC functions, the global functions do everything you need in a COM in-process component. The `DllRegisterServer()` function can be called by a utility program such as **regsvr32** to update the system Registry.
Once you've created the skeleton project, your next step is to use ClassWizard to add one or more COM-creatable classes to the project. Just fill in the **New Class** dialog box, as shown here.

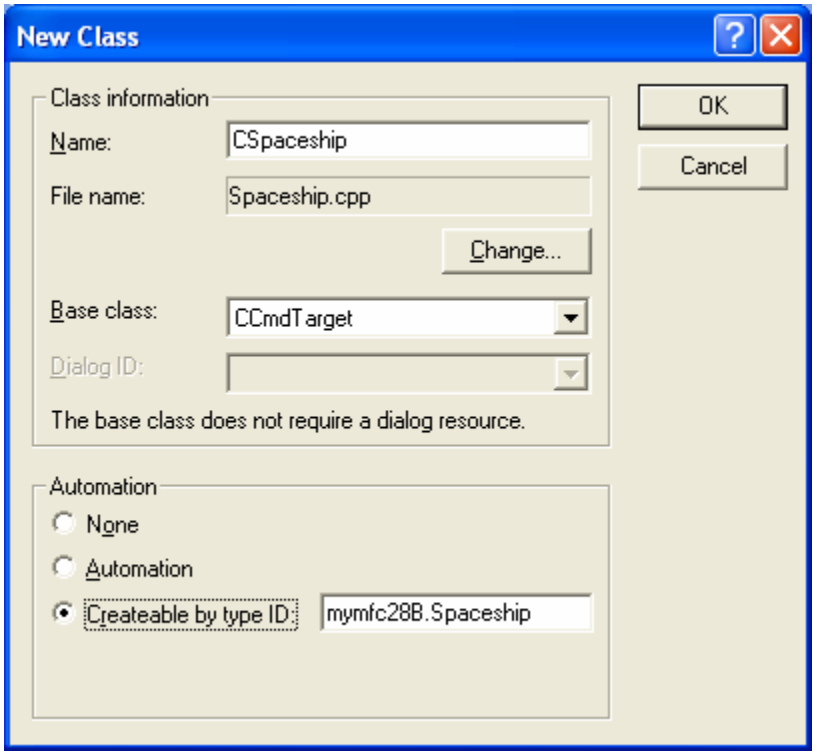

Figure 13: Adding new class to the COM project.

In your generated class, you end up with some **Automation** elements such as dispatch maps, but you can safely remove those. You can also remove (or commented out) the following two lines from **StdAfx.h**:

```
#include <afxodlgs.h>
#include <afxdisp.h>
```

```
#ifndef _AFX_NO_OLE_SUPPORT
#include <afxole.h>          // MFC OLE classes
// #include <afxodlgs.h>       // MFC OLE dialog classes
// #include <afxdisp.h>        // MFC Automation classes
#endif // _AFX_NO_OLE_SUPPORT
```

Listing 5.

The MYMFC28B Example: An MFC COM In-Process Component

The MYMFC28B example is an **MFC regular DLL** that incorporates a true COM version of the `CSpaceship` class you saw in MYMFC28A. AppWizard generated the **mymfc28B.cpp** and **mymfc28B.h** files, as described previously. Listing 7 shows the **Interface.h** file, which declares the `IMotion` and `IVisual` interfaces. Listing 8 and 9 show the code for the `CSpaceship` class. Compare the code to the code in MYMFC28A. Do you see how the use of the MFC macros reduces code size? Note that the MFC `CCmdTarget` class takes care of the reference counting and `QueryInterface()` logic.

The steps for the MYMFC28B, MFC regular DLL program are shown below.

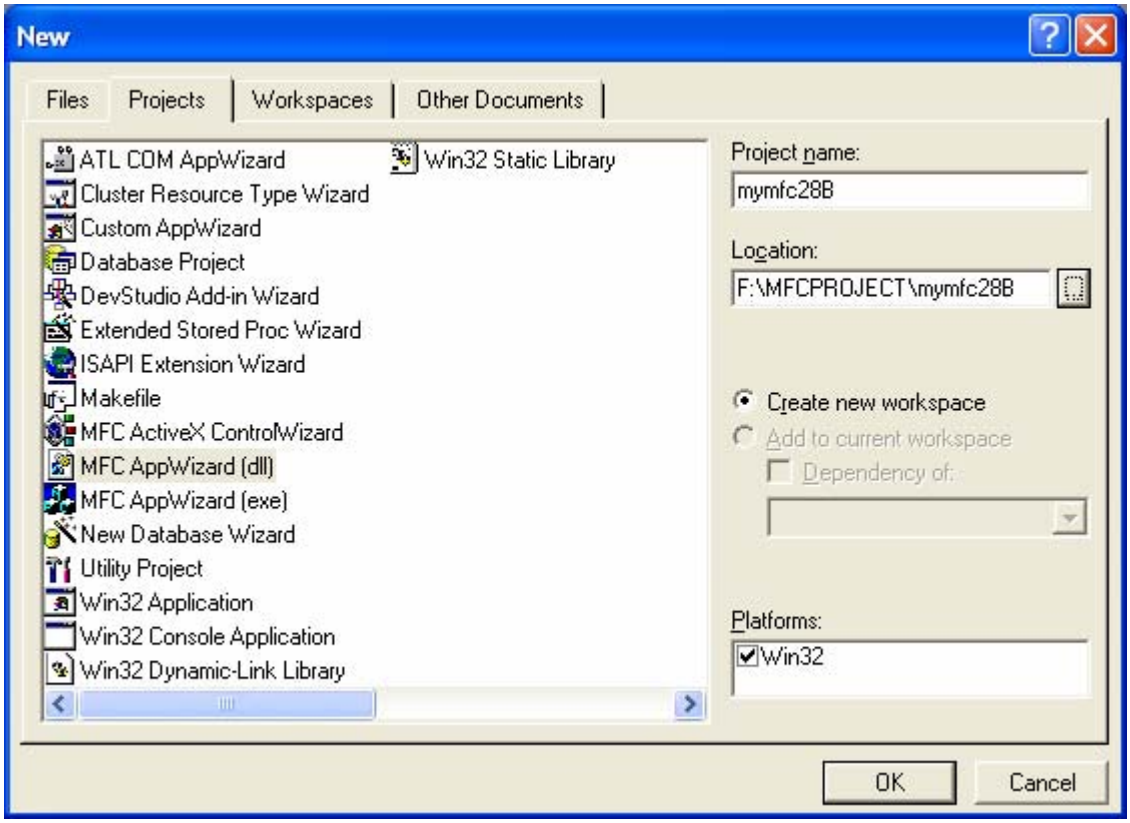

Figure 14: MYMFC28B - AppWizard new project dialog.

Don't forget to tick the **Automation** check box.

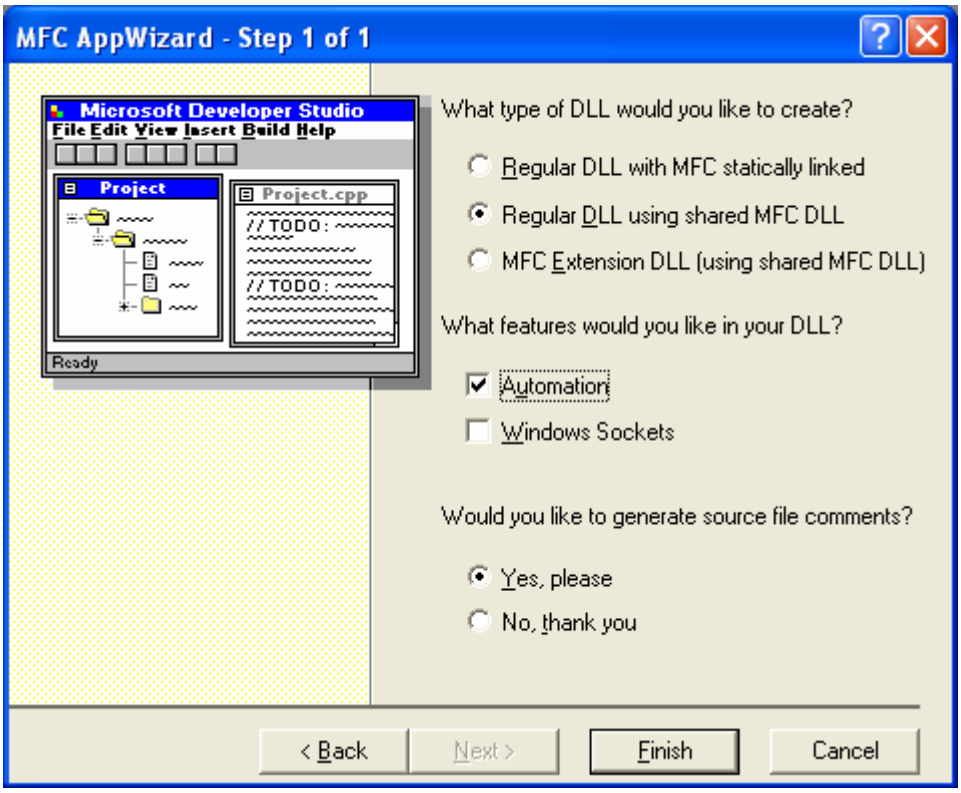

Figure 15: MYMFC28B - AppWizard DLL step 1 of 1.

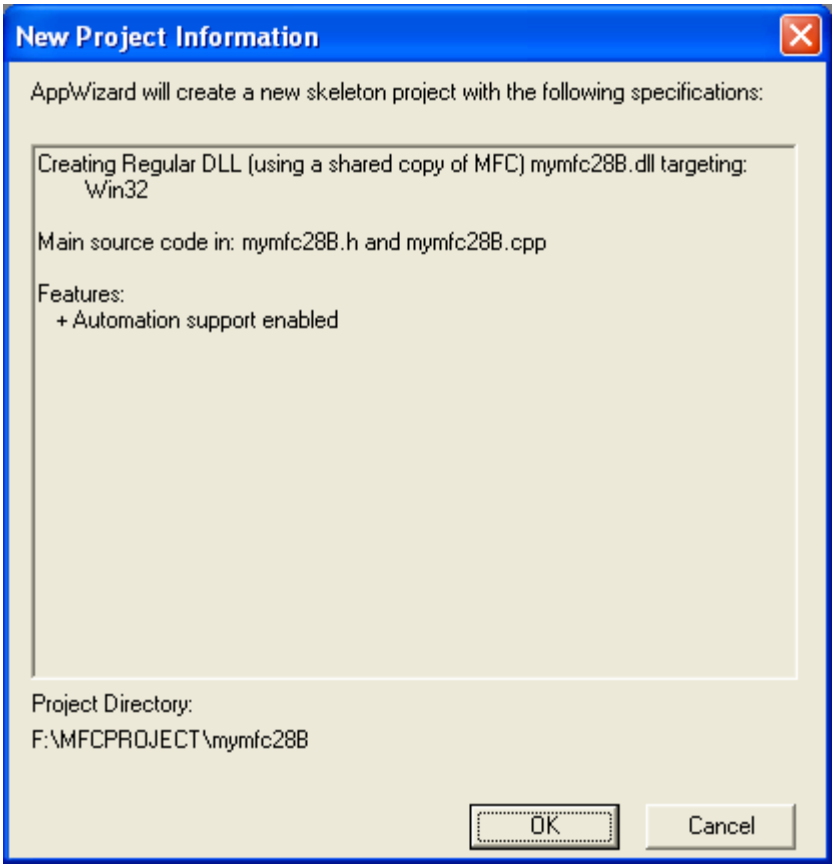

Figure 16: MYMFC28B project summary.

Add the `CSpaceship` class.

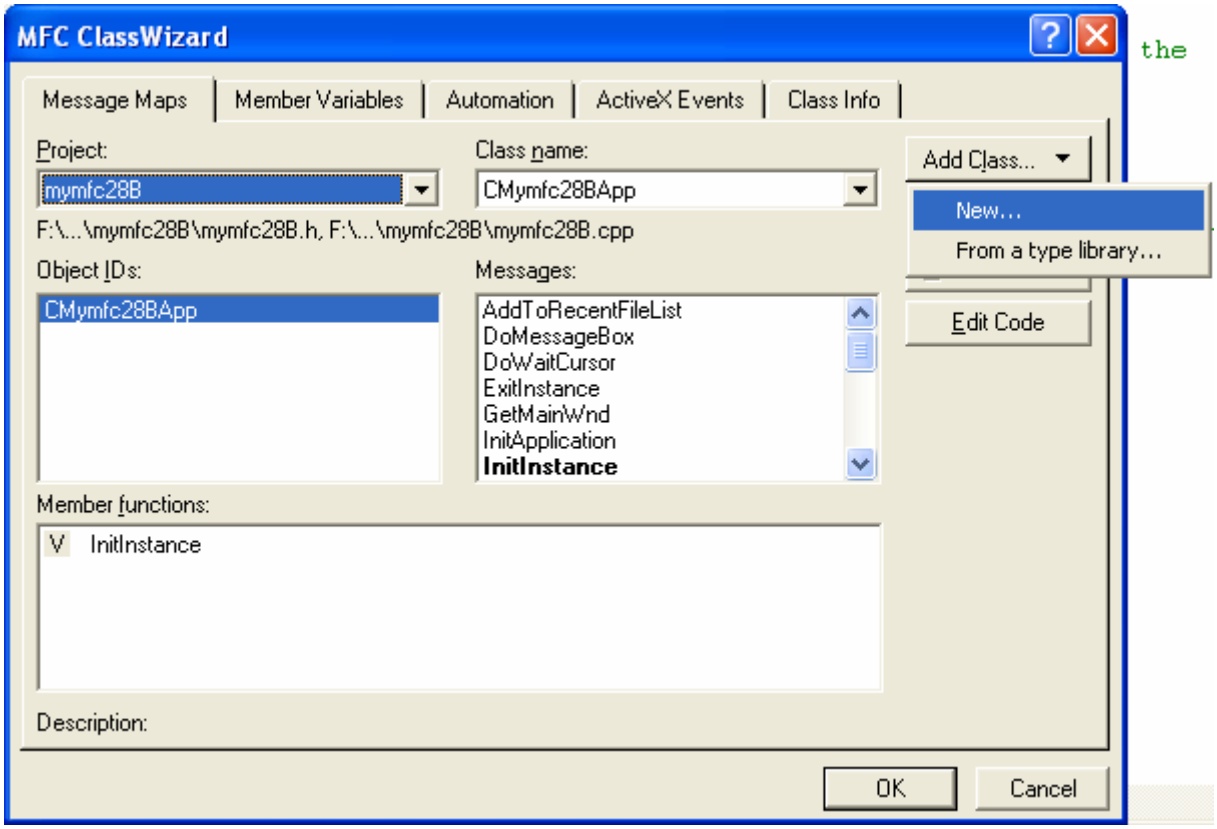

Figure 17: Adding new class to project.

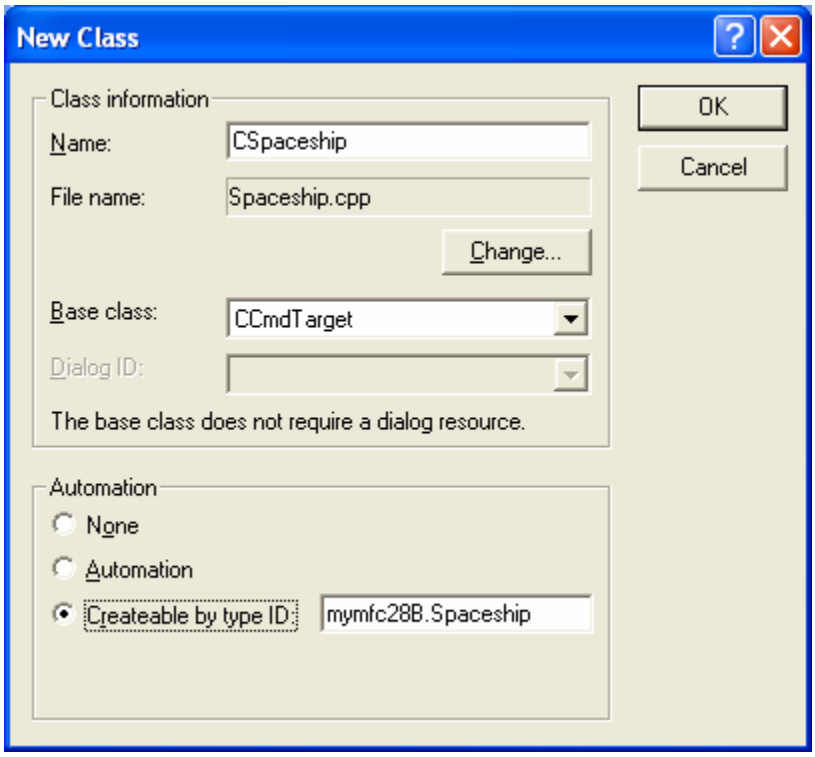

Figure 18: Entering new class information.

Add the **Interface.h** header file.

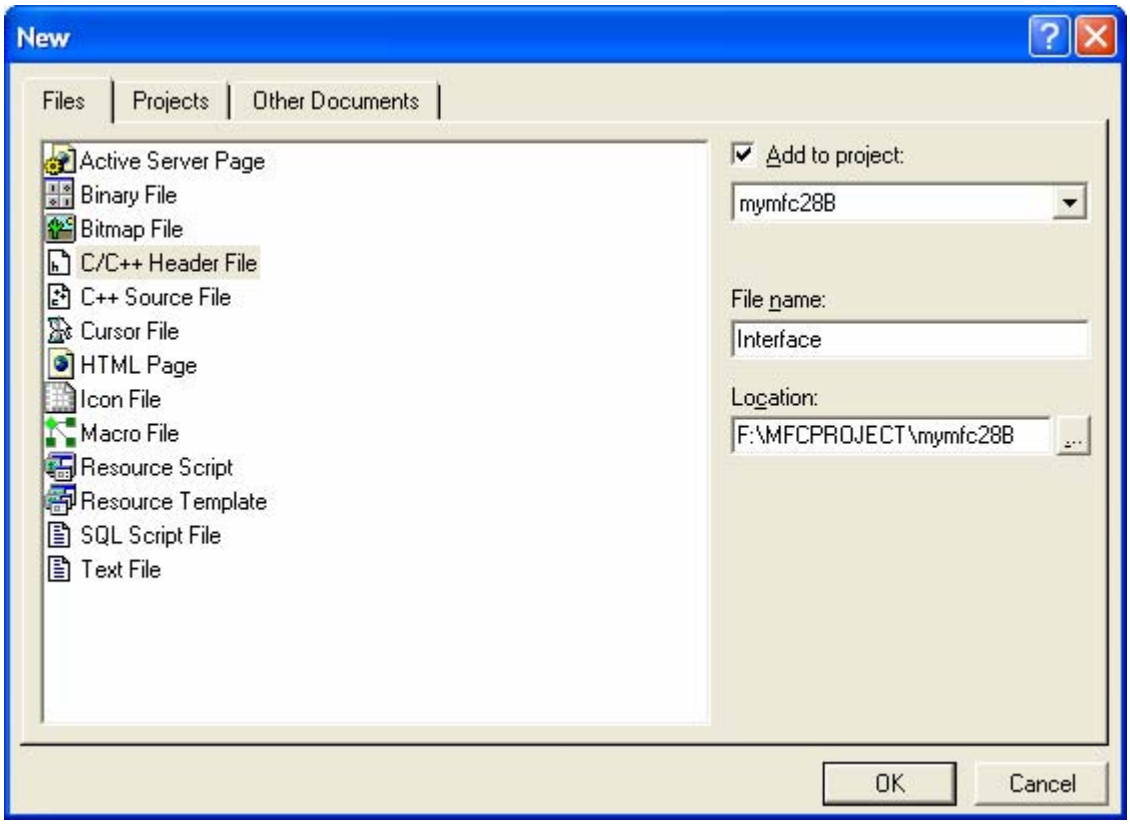

Figure 19: Adding new header file to project.

Copy the **Interface.h**, **Spaceship.h** and **Spaceship.cpp** contents as in the following Listings into the respective files in the project.
Remove or comment out the following `#include` statements.

```
#ifndef _AFX_NO_OLE_SUPPORT
#include <afxole.h>          // MFC OLE classes
#include <afxodlgs.h>        // MFC OLE dialog classes
#include <afxdisp.h>         // MFC Automation classes
#endif // _AFX_NO_OLE_SUPPORT
```

Listing 6.

Build your program, make sure no error or warning. This will generate the **mymfc28B.dll** file.

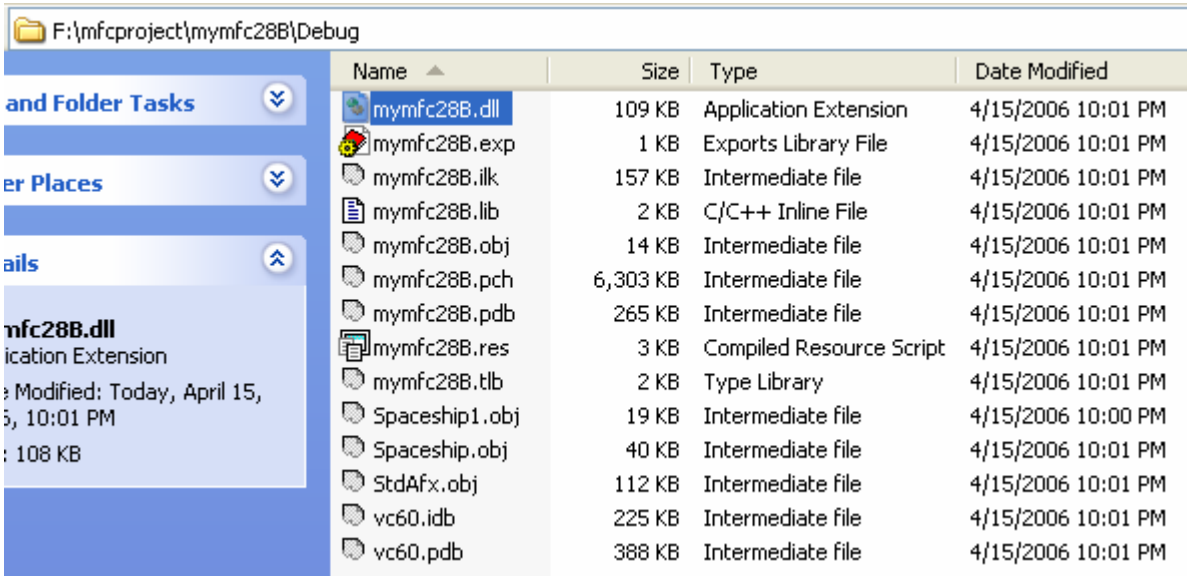

Figure 20: MYMFC28B - the generated DLL file.

Next, proceed to the client program, MYMFC28C.

INTERFACE.H

```
// interface.h

struct IMotion : public IUnknown
{
    STDMETHOD_(void, Fly) () = 0;
    STDMETHOD_(int&, GetPosition) () = 0;
};

struct IVisual : public IUnknown
{
    STDMETHOD_(void, Display) () = 0;
};
```

Listing 7: The **Interface.h** file.

```
SPACESHIP.H

#if !defined(AFX_SPACESHIP_H__170C4C09_9012_41F7_B943_7F3B4906BA37__INCLUDED_)
#define AFX_SPACESHIP_H__170C4C09_9012_41F7_B943_7F3B4906BA37__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

void ITrace(REFIID iid, const char* str);

/////////////////////////////////////////////////////////////////////////
// CSpaceship command target

class CSpaceship : public CCmdTarget
{
    DECLARE_DYNCREATE(CSpaceship)

private:
    int m_nPosition; // We can access this from all the interfaces
    int m_nAcceleration;
    int m_nColor;
protected:
    CSpaceship();    // protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSpaceship)
    public:
    virtual void OnFinalRelease();
    //}}AFX_VIRTUAL

// Implementation
protected:
    virtual ~CSpaceship();

    // Generated message map functions
    //{{AFX_MSG(CSpaceship)
        // NOTE - the ClassWizard will add and remove member
        //  functions here.
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
    DECLARE_OLECREATE(CSpaceship)
    BEGIN_INTERFACE_PART(Motion, IMotion)
        STDMETHOD_(void, Fly) ();

        STDMETHOD_(int&, GetPosition) ();
    END_INTERFACE_PART(Motion)

    BEGIN_INTERFACE_PART(Visual, IVisual)
        STDMETHOD_(void, Display) ();
    END_INTERFACE_PART(Visual)

    DECLARE_INTERFACE_MAP()
```

```
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_SPACESHIP_H__170C4C09_9012_41F7_B943_7F3B4906BA37__INCLUDED_)
```

Listing 8: The **Spaceship.h** file.

SPACESHIP.CPP

```
// Spaceship.cpp : implementation file
//

#include "stdAfx.h"
#include "mymfc28B.h"
#include "Interface.h"
#include "Spaceship.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CSpaceship

// {692D03A4-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IMotion = { 0x692d03a4, 0xc689, 0x11ce,
    { 0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e } };

// {692D03A5-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IVisual = { 0x692d03a5, 0xc689, 0x11ce,
    { 0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e } };

IMPLEMENT_DYNCREATE(CSpaceship, CCmdTarget)
CSpaceship::CSpaceship()
{
    TRACE("CSpaceship ctor\n");
    m_nPosition = 100;
    m_nAcceleration = 101;
    m_nColor = 102;
    // To keep the application running as long as an OLE automation
    //  object is active, the constructor calls AfxOleLockApp.

    AfxOleLockApp();
}

CSpaceship::~CSpaceship()
{
    TRACE("CSpaceship dtor\n");
    // To terminate the application when all objects created with
    //  OLE automation, the destructor calls AfxOleUnlockApp.

    AfxOleUnlockApp();
}
void CSpaceship::OnFinalRelease()
{
    // When the last reference for an automation object is released
    //  OnFinalRelease is called. This implementation deletes the
```

```
    //  object. Add additional cleanup required for your object before
    //  deleting it from memory.

    delete this;
}
BEGIN_MESSAGE_MAP(CSpaceship, CCmdTarget)
    //{{AFX_MSG_MAP(CSpaceship)
    // NOTE - ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

BEGIN_INTERFACE_MAP(CSpaceship, CCmdTarget)
    INTERFACE_PART(CSpaceship, IID_IMotion, Motion)
    INTERFACE_PART(CSpaceship, IID_IVisual, Visual)
END_INTERFACE_MAP()

// {692D03A3-C689-11CE-B337-88EA36DE9E4E}
IMPLEMENT_OLECREATE(CSpaceship, "Spaceship", 0x692d03a3, 0xc689,
                    0x11ce, 0xb3, 0x37, 0x88, 0xea, 0x36, 0xde,
                    0x9e, 0x4e)
STDMETHODIMP_(ULONG) CSpaceship::XMotion::AddRef()
{
    TRACE("CSpaceship::XMotion::AddRef\n");
    METHOD_PROLOGUE(CSpaceship, Motion)
    return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CSpaceship::XMotion::Release()
{
    TRACE("CSpaceship::XMotion::Release\n");
    METHOD_PROLOGUE(CSpaceship, Motion)
    return pThis->ExternalRelease();
}

STDMETHODIMP CSpaceship::XMotion::QueryInterface(
    REFIID iid, LPVOID* ppvObj)
{
    ITrace(iid, "CSpaceship::XMotion::QueryInterface");
    METHOD_PROLOGUE(CSpaceship, Motion)
    return pThis->ExternalQueryInterface(&iid, ppvObj);
}
STDMETHODIMP_(void) CSpaceship::XMotion::Fly()
{
    TRACE("CSpaceship::XMotion::Fly\n");
    METHOD_PROLOGUE(CSpaceship, Motion)
    TRACE("m_nPosition = %d\n", pThis->m_nPosition);
    TRACE("m_nAcceleration = %d\n", pThis->m_nAcceleration);
    return;
}

STDMETHODIMP_(int&) CSpaceship::XMotion::GetPosition()
{
    TRACE("CSpaceship::XMotion::GetPosition\n");
    METHOD_PROLOGUE(CSpaceship, Motion)
    TRACE("m_nPosition = %d\n", pThis->m_nPosition);
    TRACE("m_nAcceleration = %d\n", pThis->m_nAcceleration);
    return pThis->m_nPosition;
}

/////////////////////////////////////////////////////////////////////////////
STDMETHODIMP_(ULONG) CSpaceship::XVisual::AddRef()
{
    TRACE("CSpaceship::XVisual::AddRef\n");
```

```
    METHOD_PROLOGUE(CSpaceship, Visual)
    return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CSpaceship::XVisual::Release()
{
    TRACE("CSpaceship::XVisual::Release\n");
    METHOD_PROLOGUE(CSpaceship, Visual)
    return pThis->ExternalRelease();
}

STDMETHODIMP CSpaceship::XVisual::QueryInterface(
    REFIID iid, LPVOID* ppvObj)
{
    ITrace(iid, "CSpaceship::XVisual::QueryInterface");

    METHOD_PROLOGUE(CSpaceship, Visual)
    return pThis->ExternalQueryInterface(&iid, ppvObj);
}
STDMETHODIMP_(void) CSpaceship::XVisual::Display()
{
    TRACE("CSpaceship::XVisual::Display\n");
    METHOD_PROLOGUE(CSpaceship, Visual)
    TRACE("m_nPosition = %d\n", pThis->m_nPosition);
    TRACE("m_nColor = %d\n", pThis->m_nColor);
}

/////////////////////////////////////////////////////////////////////////
void ITrace(REFIID iid, const char* str)
{
    OLECHAR* lpszIID;
    ::StringFromIID(iid, &lpszIID);
    CString strTemp = (LPCWSTR) lpszIID;
    TRACE("%s - %s\n", (const char*) strTemp, (const char*) str);
    AfxFreeTaskMem(lpszIID);
}

/////////////////////////////////////////////////////////////////////////
// CSpaceship message handlers
```

Listing 9: The **Spaceship.cpp** file.

MFC COM Client Programs

Basically, writing an MFC COM client program is a no-brainer. You just use AppWizard to generate a normal application. Add the following line in **StdAfx.h**.

```
#include <afxole.h>
```

```
#include <afxdtctl.h>          // mf
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>            // MF
#endif // _AFX_NO_AFXCMN_SUPPORT

#include <afxole.h>
```

Listing 10.

Then add the following line at the beginning of the application class `InitInstance()` member function.

```
    AfxOleInit();
```

```
// CMymfc28CApp initialization

BOOL CMymfc28CApp::InitInstance()
{
    AfxOleInit();
    AfxEnableControlContainer();

    // Standard initialization
```

Listing 11.

You're now ready to add code that calls `CoGetClassObject()`.

The MYMFC28C Example: An MFC COM Client

The MYMFC28C example is an MFC program that incorporates a true COM version of the client code you saw in MYMFC28A. This is a generic AppWizard MFC **Single Document Interface** (SDI) EXE program with an added `#include` statement for the MFC COM headers and a call to `AfxOleInit()`, which initializes the DLL. A **Spaceship** option on an added **Test** menu is mapped to the view class handler function, `OnTestSpaceship()`. The project also contains a copy of the MYMFC28B component's **Interface.h** file, shown in Listing 7. You can see an `#include` statement for this file at the top of **mymfc28CView.cpp**.

The following are MYMFC28C steps.

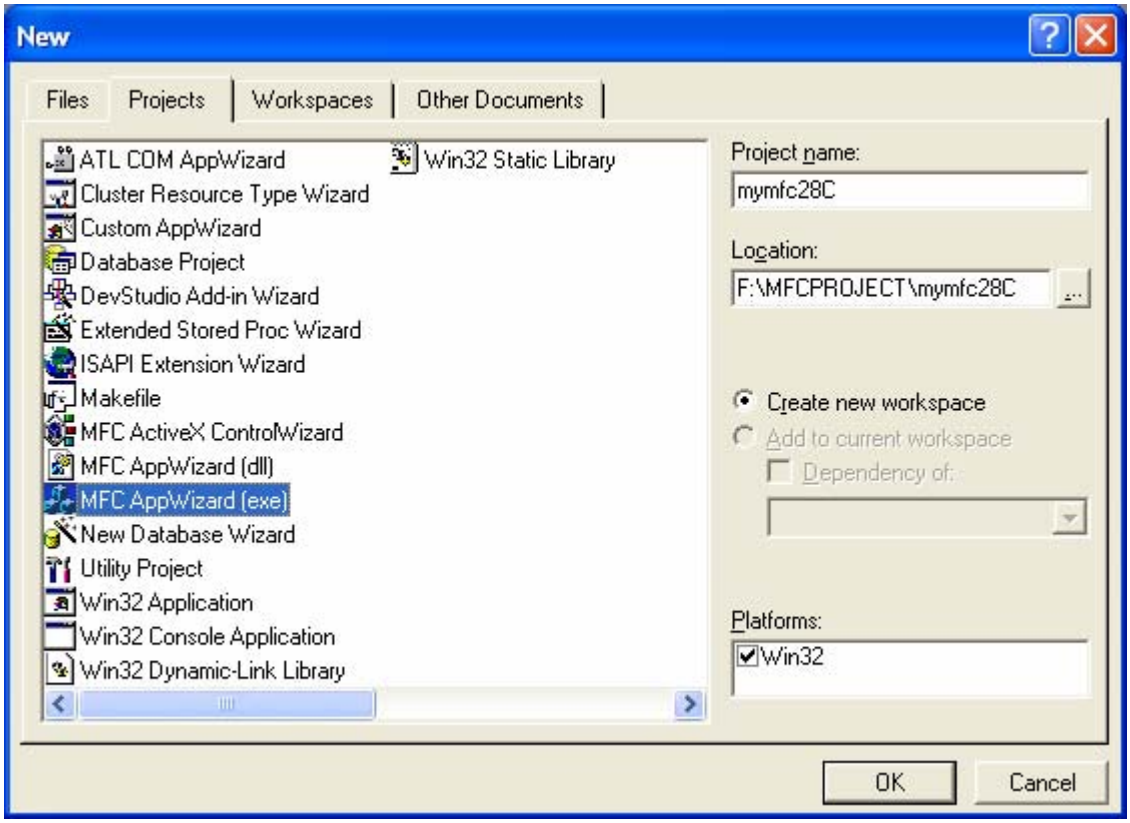

Figure 21: MYMFC28C – New EXE project dialog.

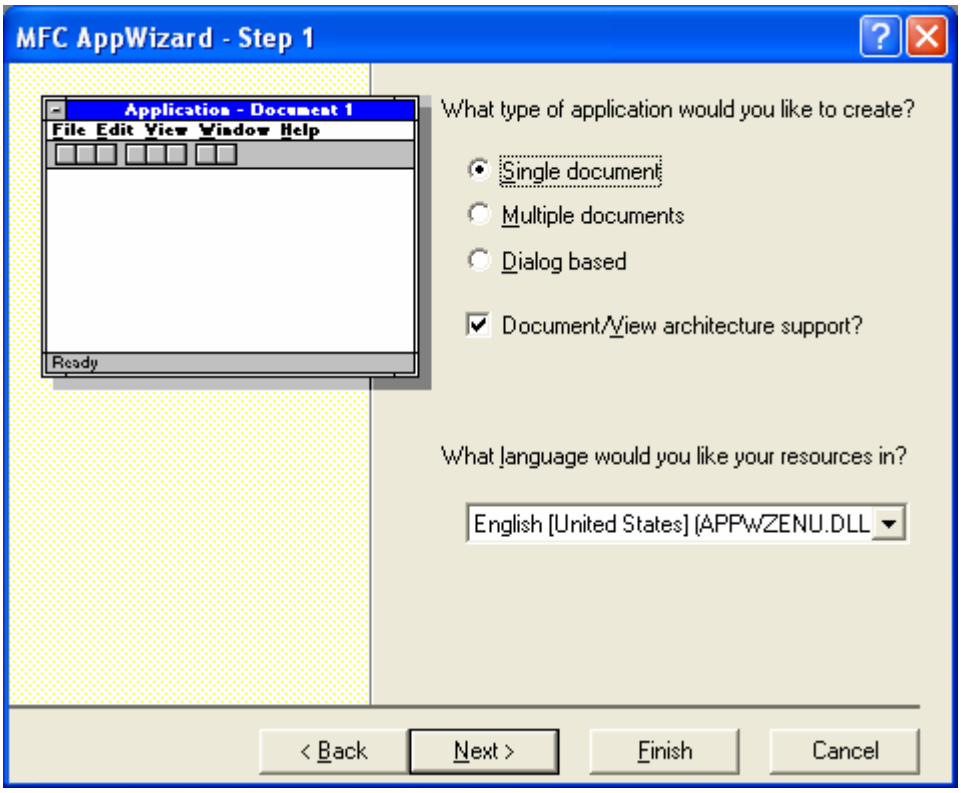

Figure 22: MYMFC28C – AppWizard step 1 of 6.

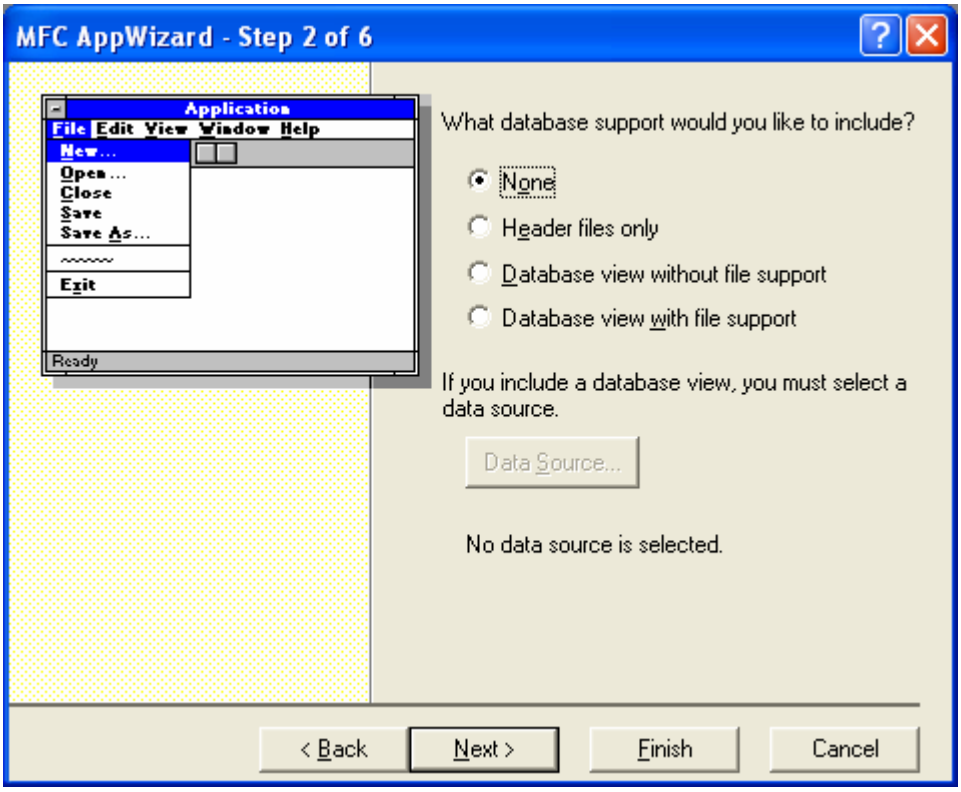

Figure 23: MYMFC28C – AppWizard step 2 of 6.

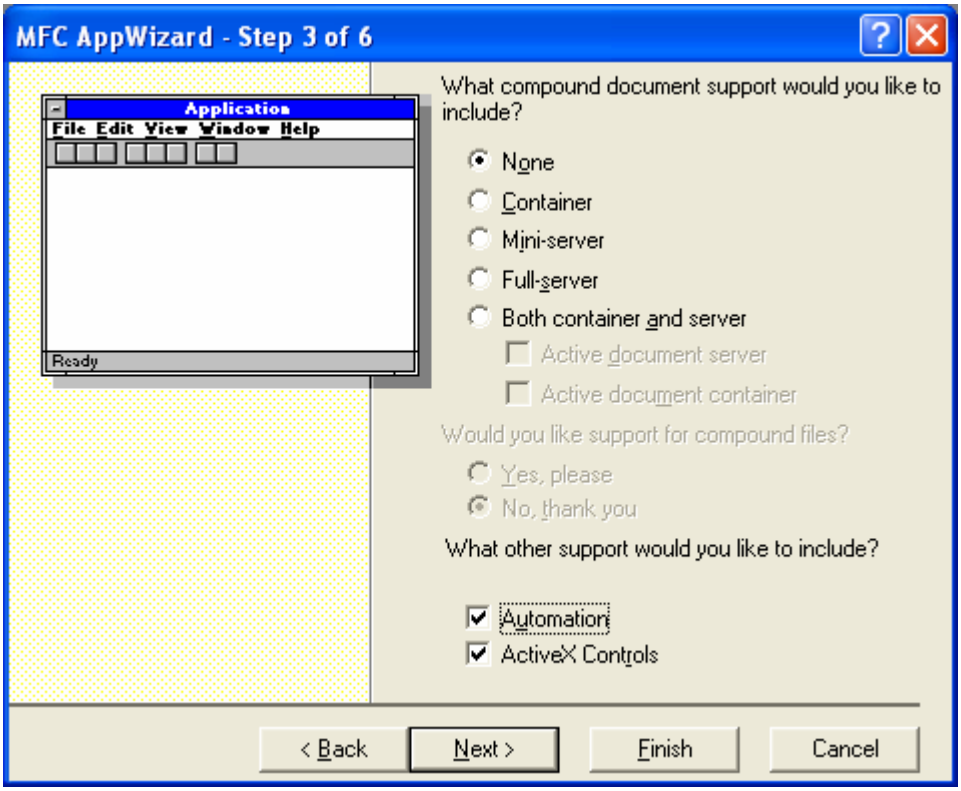

Figure 24: MYMFC28C – AppWizard step 3 of 6.

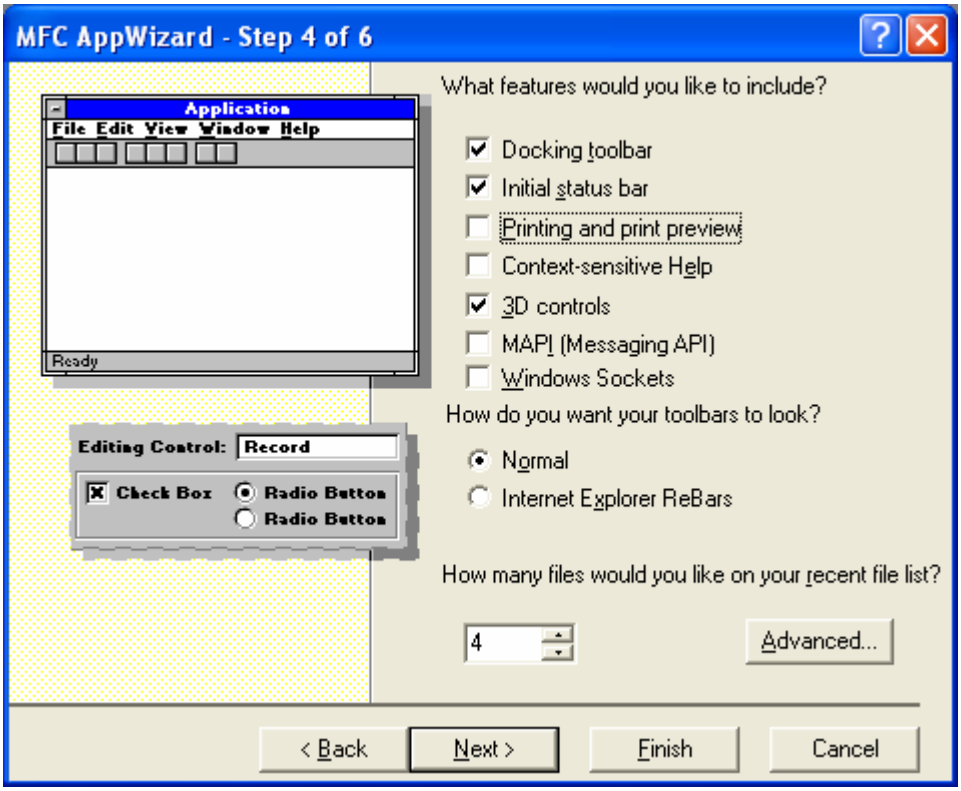

Figure 25: MYMFC28C – AppWizard step 4 of 6.

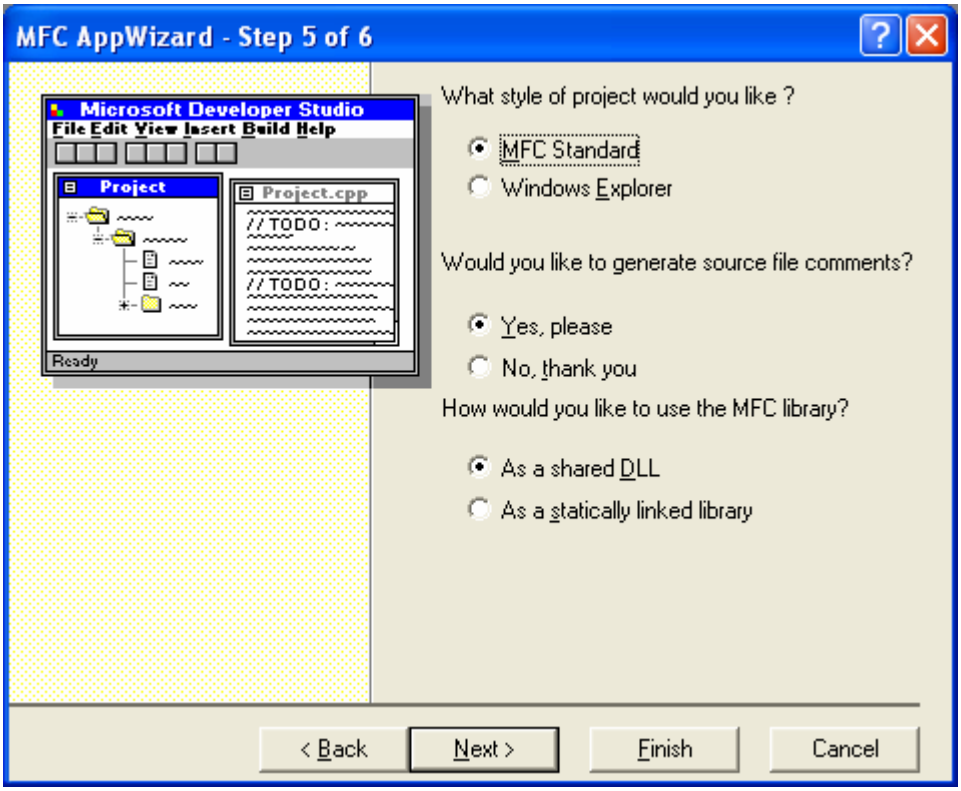

Figure 26: MYMFC28C – AppWizard step 5 of 6.

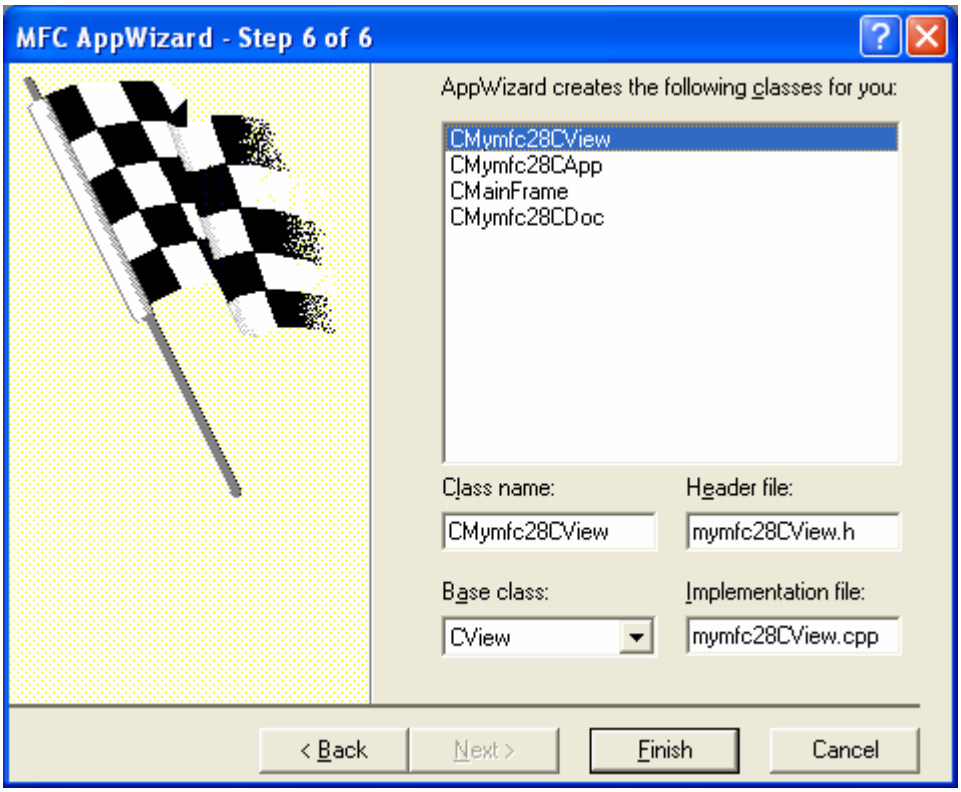

Figure 27: MYMFC28C – AppWizard step 1 of 6.

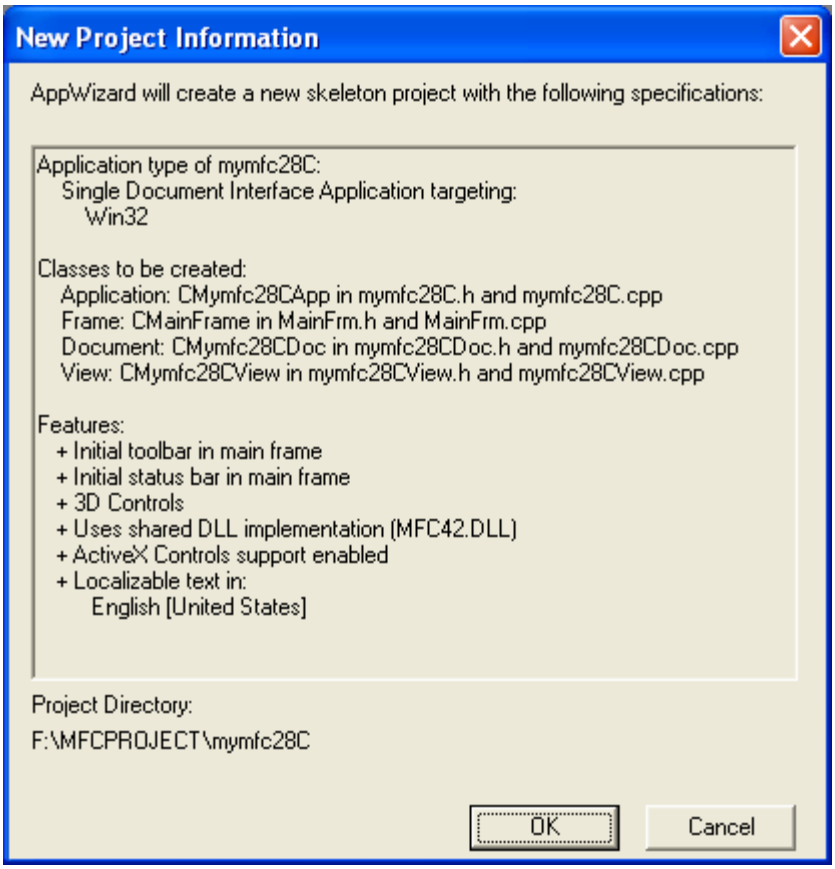

Figure 28: MYMFC28C – AppWizard project summary.

Add the following line in **StdAfx.h**:

```
#include <afxole.h>
```

```
#include <afxdtctl.h>          // mr
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>            // MF
#endif // _AFX_NO_AFXCMN_SUPPORT

#include <afxole.h>
```

Listing 12.

Then add the following line at the beginning of the application class `InitInstance()` member function:

```
AfxOleInit();
```

```
// CMymfc28CApp initialization

BOOL CMymfc28CApp::InitInstance()
{
    AfxOleInit();
    AfxEnableControlContainer();

    // Standard initialization
```

Listing 13.

Adding the **Test Spaceship** menu item for the testing purpose.

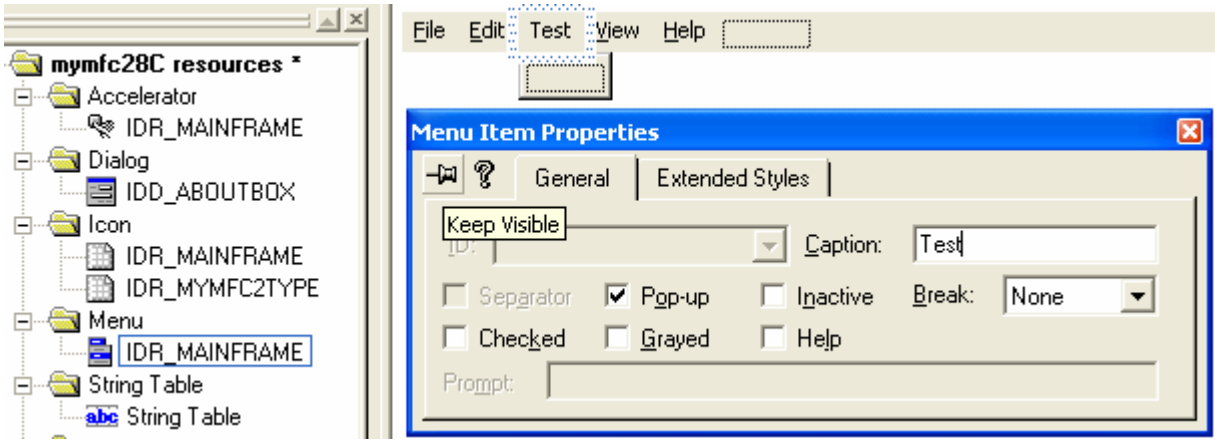

Figure 29: Adding **Test** menu.

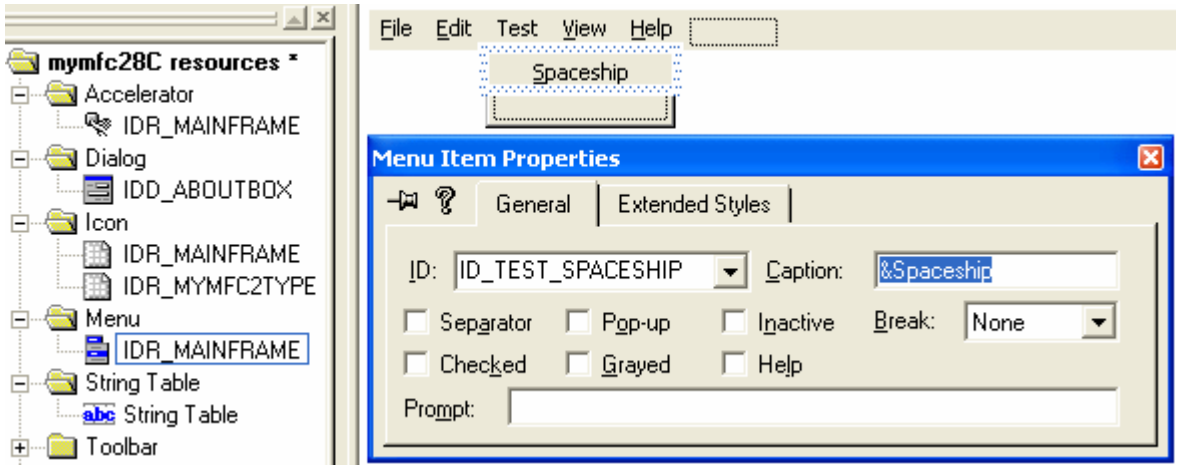

Figure 30: Adding **Spaceship** menu item.

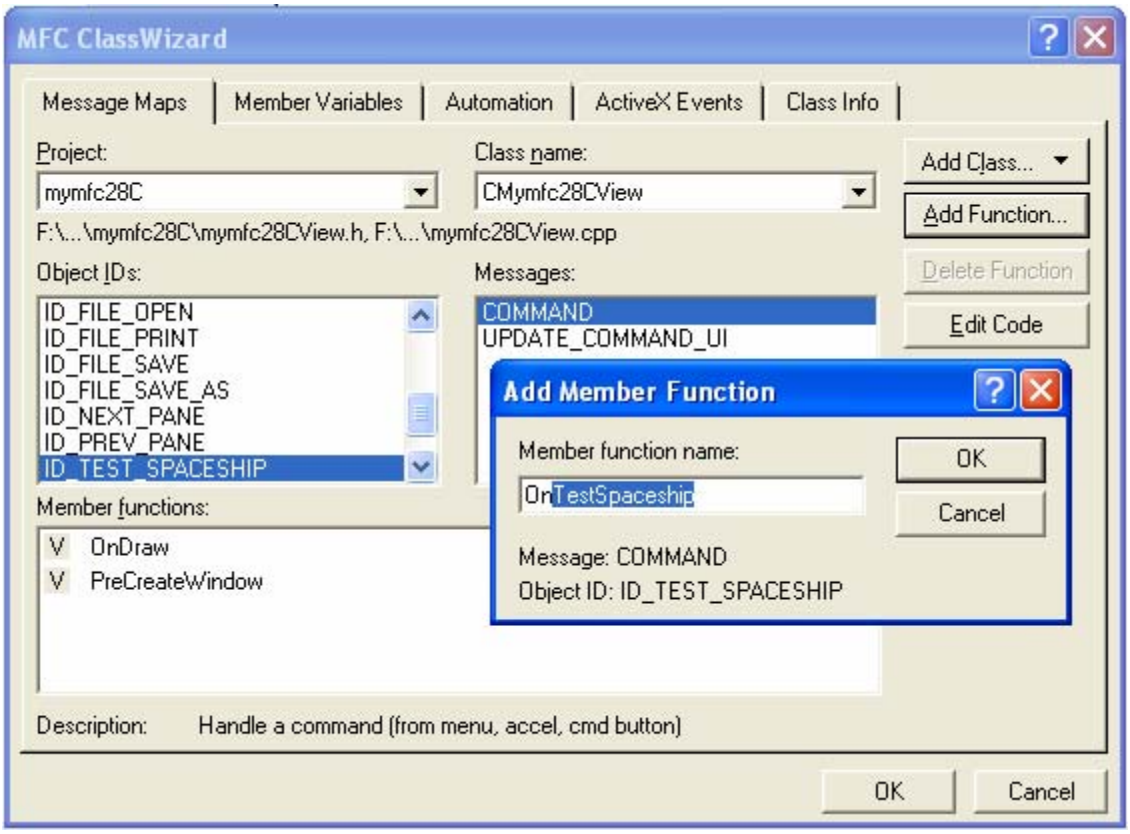

Figure 31: Adding message handler.

Adding the code that calls `CoGetClassObject()`. Click the **Edit Code** button and add the following code for `CMymfc28CView::OnTestSpaceship()`.

```
void CMymfc28CView::OnTestSpaceship()
{
    // TODO: Add your command handler code here
    CLSID clsid;
    LPCLASSFACTORY pClf;
    LPUNKNOWN pUnk;
    IMotion* pMot;
    IVisual* pVis;

    HRESULT hr;
    if ((hr = ::CLSIDFromProgID(L"Spaceship", &clsid)) != NOERROR)
    {
        TRACE("unable to find Program ID -- error = %x\n", hr);
        return;
    }
    if ((hr = ::CoGetClassObject(clsid, CLSCTX_INPROC_SERVER, NULL,
IID_IClassFactory, (void **) &pClf)) != NOERROR)
    {
        TRACE("unable to find CLSID -- error = %x\n", hr);
        return;
    }

    pClf->CreateInstance(NULL, IID_IUnknown, (void**) &pUnk);
    pUnk->QueryInterface(IID_IMotion, (void**) &pMot); // All three
    pMot->QueryInterface(IID_IVisual, (void**) &pVis); //  pointers
                                                       //  should work
```

```
        TRACE("main: pUnk = %p, pMot = %p, pDis = %p\n", pUnk, pMot, pVis);

        // Test all the interface virtual functions
        pMot->Fly();
        int nPos = pMot->GetPosition();
        TRACE("nPos = %d\n", nPos);
        pVis->Display();

        pClf->Release();
        pUnk->Release();
        pMot->Release();
        pVis->Release();
        AfxMessageBox("Test succeeded. See Debug window for output.");
    }
```

```
void CMymfc28CView::OnTestSpaceship()
{
    // TODO: Add your command handler code here
    CLSID clsid;
    LPCLASSFACTORY pClf;
    LPUNKNOWN pUnk;
    IMotion* pMot;
    IVisual* pVis;

    HRESULT hr;
    if ((hr = ::CLSIDFromProgID(L"Spaceship", &clsid)) != NOERROR)
    {
        TRACE("unable to find Program ID -- error = %x\n", hr);
        return;
    }
    if ((hr = ::CoGetClassObject(clsid, CLSCTX_INPROC_SERVER, NULL, IID_IClassFactory,
        (void **) &pClf)) != NOERROR)
    {
        TRACE("unable to find CLSID -- error = %x\n", hr);
        return;
    }

    pClf->CreateInstance(NULL, IID_IUnknown, (void**) &pUnk);
    pUnk->QueryInterface(IID_IMotion, (void**) &pMot); // All three
    pMot->QueryInterface(IID_IVisual, (void**) &pVis); //  pointers
                                                       //  should work
    TRACE("main: pUnk = %p, pMot = %p, pDis = %p\n", pUnk, pMot, pVis);

    // Test all the interface virtual functions
    pMot->Fly();
    int nPos = pMot->GetPosition();
    TRACE("nPos = %d\n", nPos);
    pVis->Display();

    pClf->Release();
    pUnk->Release();
    pMot->Release();
    pVis->Release();
    AfxMessageBox("Test succeeded. See Debug window for output.");
}
```

Listing 14.

Define the `GUID`s for our custom classes and interfaces of the spaceship objects in the **CMymfc28CView.cpp** at the top of the class implementation.

```
// {692D03A4-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IMotion =
{ 0x692d03a4, 0xc689, 0x11ce, { 0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e } };
```

```
// {692D03A5-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IVisual =
{ 0x692d03a5, 0xc689, 0x11ce, { 0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e } };
```

```
#endif

// {692D03A4-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IMotion =
{ 0x692d03a4, 0xc689, 0x11ce, { 0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e } };

// {692D03A5-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IVisual =
{ 0x692d03a5, 0xc689, 0x11ce, { 0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e } };

/////////////////////////////////////////////////////////////////////////////
// CMymfc28CView
```

Listing 15.

Copy (not add to the project) the **Interface.h** file from MYMFC28B project into the MYMFC28C project directory. Put the `#include` statement in the **CMymfc28CView.cpp**.

```
#include "stdafx.h"
#include "mymfc28C.h"

#include "mymfc28CDoc.h"
#include "mymfc28CView.h"
#include "Interface.h"

#ifdef _DEBUG
```

Listing 16.

Register the previous **mymfc28B.dll** using the **regsvr32** (**regsvr** for older Windows OS) utility at command prompt.

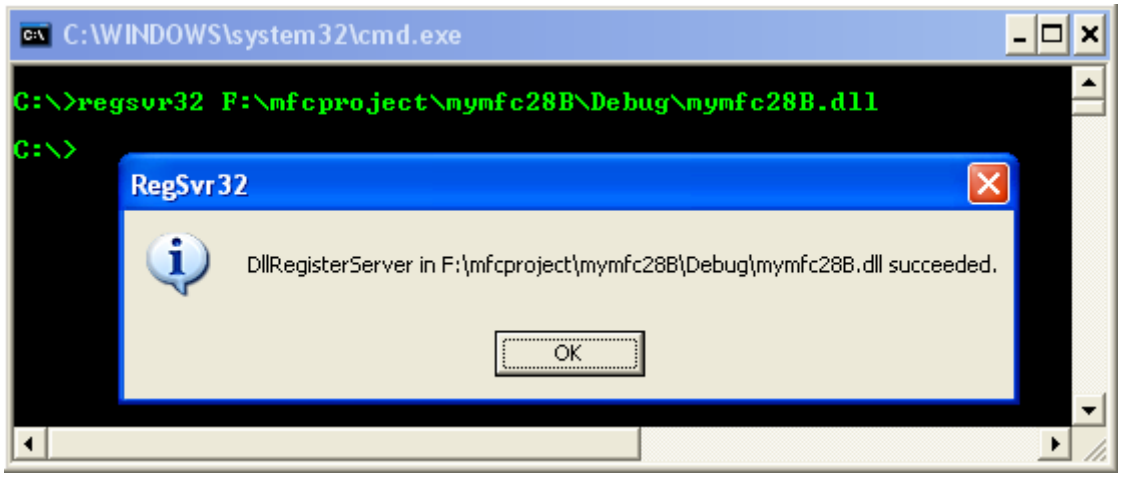

Figure 32: Registering DLL using **regsvr32**.

Build and run MYMFC28C program through the debugger. Select the **Test Spaceship** menu and see the TRACE in the Debug Window.

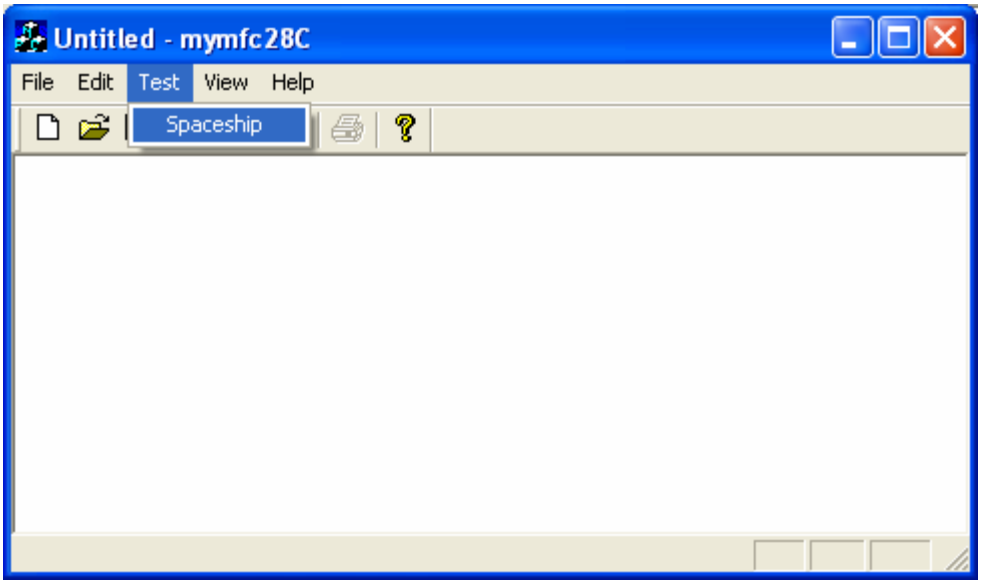

Figure 33: MYMFC28C output.

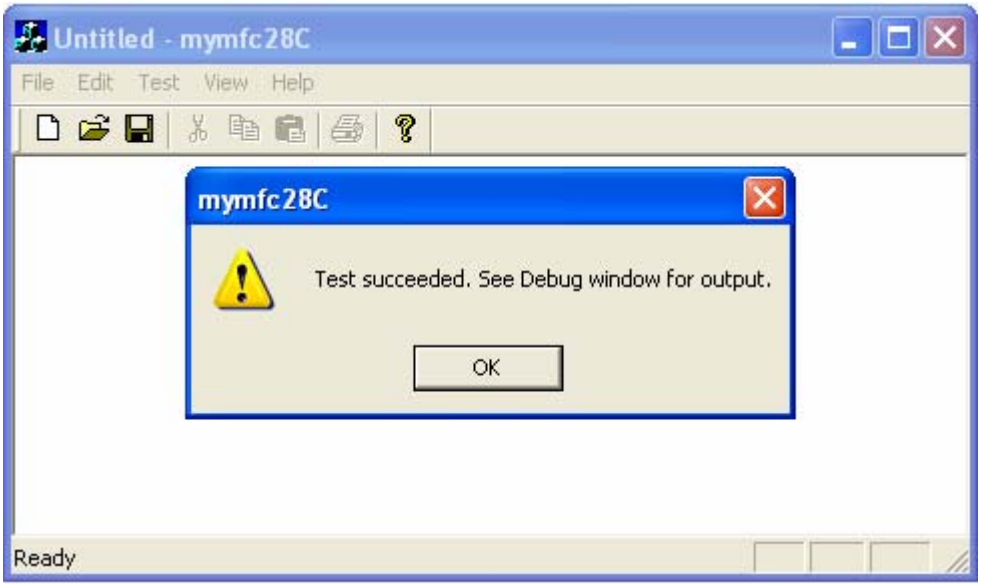

Figure 34: MYMFC28C in action.

The following messages seen through the Debug Window.

```
...
...
Loaded symbols for 'F:\mfcproject\mymfc28B\Debug\mymfc28B.dll'
CSpaceship ctor
{692D03A4-C689-11CE-B337-88EA36DE9E4E} - CSpaceship::XMotion::QueryInterface
{692D03A5-C689-11CE-B337-88EA36DE9E4E} - CSpaceship::XMotion::QueryInterface
main: pUnk = 00423FEC, pMot = 00423FEC, pDis = 00423FF0
CSpaceship::XMotion::Fly
m_nPosition = 100
m_nAcceleration = 101
CSpaceship::XMotion::GetPosition
m_nPosition = 100
```

```
m_nAcceleration = 101
nPos = 100
CSpaceship::XVisual::Display
m_nPosition = 100
m_nColor = 102
CSpaceship::XMotion::Release
CSpaceship::XMotion::Release
CSpaceship::XVisual::Release
CSpaceship dtor
Info: AfxDllCanUnloadNow returning S_OK
The thread 0x9DC has exited with code 0 (0x0).
The program 'F:\mfcproject\mymfc28C\Debug\mymfc28C.exe' has exited with code 0 (0x0).
```

To test the client and the component, you must first run the component to update the Registry. Several utilities such as **regsvr32** can be used to do this. Both client and component show their progress through TRACE calls, so you need the debugger. You can run either the client or the component from the debugger. If you try to run the **component**, you'll be prompted for the **client** pathname as shown below.

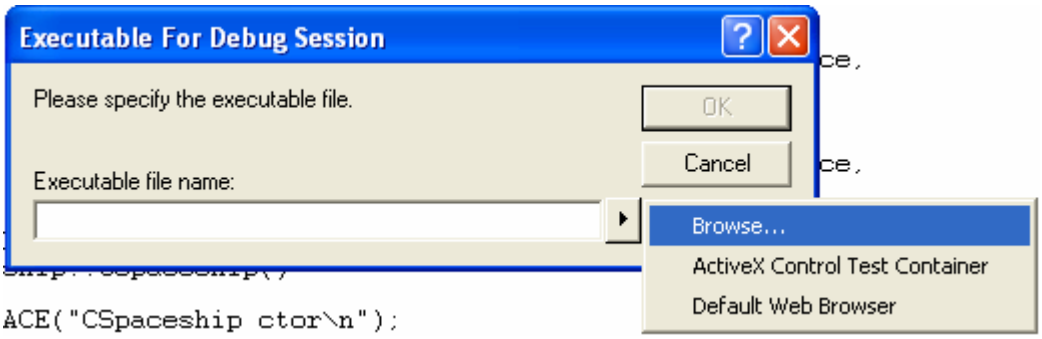

Figure 35: Client program is required when running the component standalone.

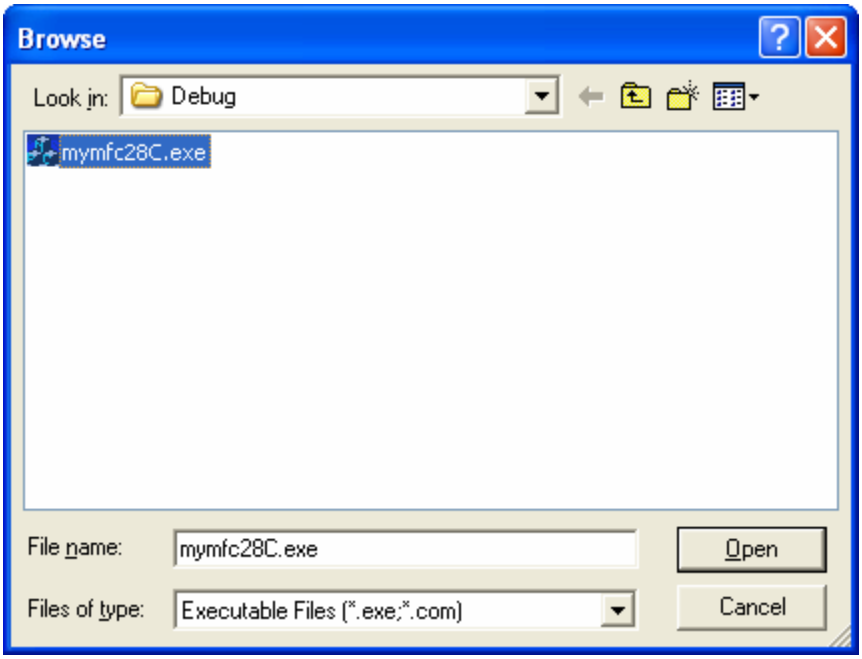

Figure 36: Browsing and selecting the client program.

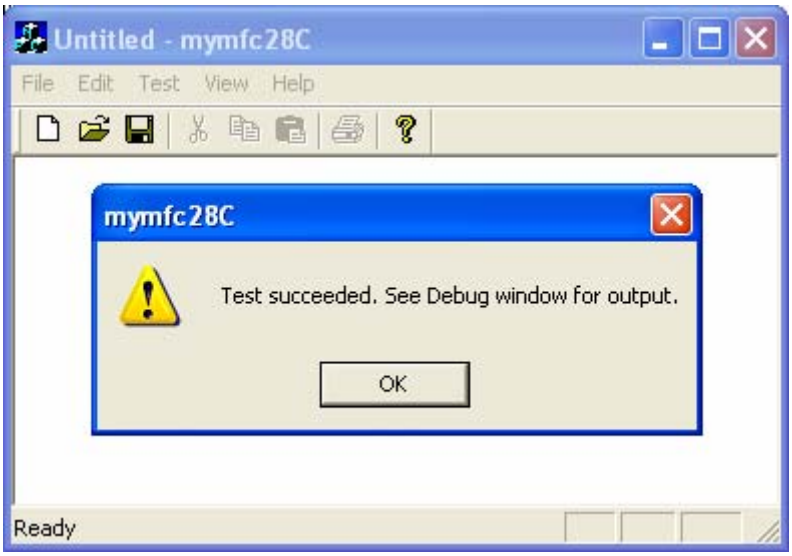

Figure 37: MYMFC28C in action.

In either case, you don't have to copy the DLL because Windows finds it through the Registry.

Containment and Aggregation vs. Inheritance

In normal C++ programming, you frequently use inheritance to factor out common behavior into a reusable base class. The `CPersistentFrame` class (discussed in Module 9) is an example of reusability through inheritance. COM uses **containment** and **aggregation** instead of **inheritance**. Let's start with containment. Suppose you extended the spaceship simulation to include planets in addition to spaceships. Using C++ by itself, you would probably write a `COrbiter` base class that encapsulated the laws of planetary motion. With COM, you would have "outer" `CSpaceship` and `CPlanet` classes plus an "inner" `COrbiter` class. The outer classes would implement the `IVisual` interface directly, but those outer classes would delegate their `IMotion` interfaces to the inner class. The result would look something like this.

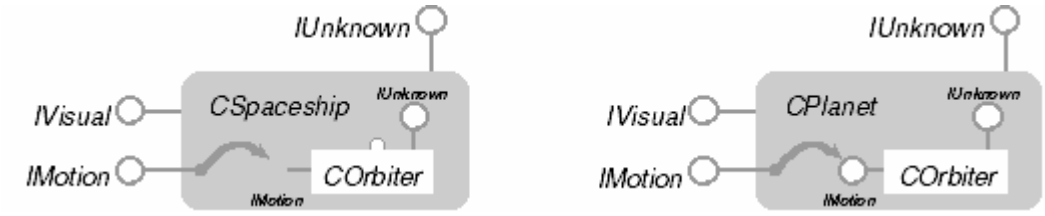

Figure 38: COM Containment.

Note that the `COrbiter` object doesn't know that it's inside a `CSpaceship` or `CPlanet` object, but the outer object certainly knows that it has a `COrbiter` object embedded inside. The outer class needs to implement all its interface functions, but the `IMotion` functions, including `QueryInterface()`, simply call the same `IMotion` functions of the inner class.
A more complex alternative to containment is **aggregation**. With aggregation, the client can have direct access to the inner object's interfaces. Shown here is the aggregation version of the space simulation.

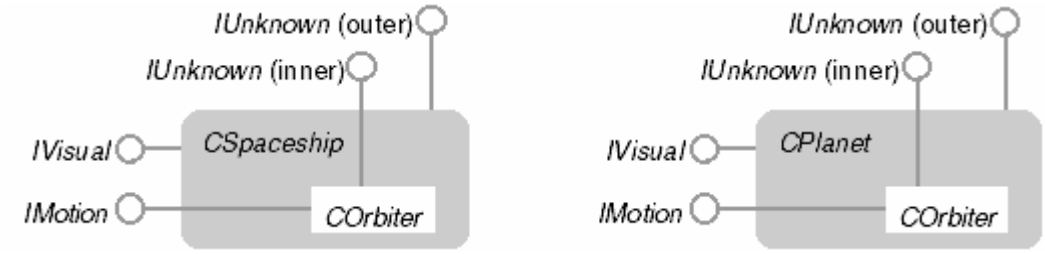

Figure 39: COM aggregation.

The orbiter is embedded in the spaceship and planet, just as it was in the containment case. Suppose the client obtains an `IVisual` pointer for a spaceship and then calls `QueryInterface()` for an `IMotion` pointer. Using the outer `IUnknown` pointer will draw a blank because the `CSpaceship` class doesn't support `IMotion`. The `CSpaceship` class keeps track of the inner `IUnknown` pointer (of its embedded `COrbiter` object), so the class uses that pointer to obtain the `IMotion` pointer for the `COrbiter` object.
Now suppose the client obtains an `IMotion` pointer and then calls `QueryInterface()` for `IVisual`. The inner object must be able to navigate to the outer object, but how? Take a close look at the `CreateInstance()` call back in Figure 24-10. The first parameter is set to NULL in that case. If you are creating an aggregated (inner) object, you use that parameter to pass an `IUnknown` pointer for the outer object that you have already created. This pointer is called the controlling unknown. The `COrbiter` class saves this pointer in a data member and then uses it to call `QueryInterface()` for interfaces that the class itself doesn't support.
The MFC library supports aggregation. The `CCmdTarget` class has a public data member `m_pOuterUnknown` that holds the outer object's `IUnknown` pointer (if the object is aggregated). The `CCmdTarget` member functions `ExternalQueryInterface()`, `ExternalAddRef()`, and `ExternalRelease()` delegate to the outer `IUnknown` if it exists. Member functions `InternalQueryInterface()`, `InternalAddRef()`, and `InternalRelease()` do not delegate. See Technical Note #38 in the online documentation for a description of the MFC macros that support aggregation.

The IUnknown Interface info

As discussed previously, the `IUnknown` interface defines three member functions that must be implemented for each object that is exposed. The prototypes for these functions reside in the header file, **Ole2.h**.

- `QueryInterface()` - Identifies which OLE interfaces the object supports.
- `AddRef()` - Increments a member variable that tracks the number of references to the object.
- `Release()` - Decrements the member variable that tracks the instances of the object. If an object has zero references, Release frees the object.

These functions provide the fundamental interface through which OLE can access objects. The following Tables are the information summary of those functions.

Item	Description
Function	`IUnknown::QueryInterface`
Use	Returns a pointer to a specified interface on an object to which a client currently holds an interface pointer. This function must call `IUnknown::AddRef` on the pointer it returns.
Prototype	`HRESULT QueryInterface(REFIID iid, void ** ppvObject);`

Parameters	`iid`
	[in] Identifier of the interface being requested.
	
	`ppvObject`
	[out] Address of pointer variable that receives the interface pointer requested in riid. Upon successful return, `*ppvObject` contains the requested interface

	pointer to the object. If the object does not support the interface specified in `iid`, `*ppvObject` is set to NULL.
Return value	`S_OK` if the interface is supported, `E_NOINTERFACE` if not.
Include file	MFC.
Remark	For any one object, a specific query for the `IUnknown` interface on any of the object's interfaces must always return the same pointer value. This allows a client to determine whether two pointers point to the same component by calling QueryInterface on both and comparing the results. It is specifically not the case that queries for interfaces (even the same interface through the same pointer) must return the same pointer value. There are four requirements for implementations of `QueryInterface()` (In these cases, "must succeed" means "must succeed barring catastrophic failure."): ▪ The set of interfaces accessible on an object through `IUnknown::QueryInterface` must be static, not dynamic. This means that if a call to `QueryInterface()` for a pointer to a specified interface succeeds the first time, it must succeed again, and if it fails the first time, it must fail on all subsequent queries. ▪ It must be reflexive - if a client holds a pointer to an interface on an object, and queries for that interface, the call must succeed. ▪ It must be symmetric - if a client holding a pointer to one interface queries successfully for another, a query through the obtained pointer for the first interface must succeed. ▪ It must be transitive - if a client holding a pointer to one interface queries successfully for a second, and through that pointer queries successfully for a third interface, a query for the first interface through the pointer for the third interface must succeed. Implementations of `QueryInterface()` must never check ACLs (Access Control List). The main reason for this rule is because COM requires that an object supporting a particular interface always return success when queried for that interface. Another reason is that checking ACLs on `QueryInterface()` does not provide any real security because any client who has access to a particular interface can hand it directly to another client without any calls back to the server. Also, because COM caches interface pointers, it does not call `QueryInterface()` on the server every time a client does a query.

Table 1.

Item	Description
Function	`IUnknown::AddRef`
Use	The `IUnknown::AddRef` method increments the reference count for an interface on an object. It should be called for every new copy of a pointer to an interface on a given object.
Prototype	`ULONG AddRef(void);`
Parameters	None.
Return value	Returns an integer from 1 to n, the value of the new reference count. This information is meant to be used for diagnostic/testing purposes only, because, in certain situations, the value may be unstable.
Include file	MFC.
Remark	Objects use a reference counting mechanism to ensure that the lifetime of the object includes the lifetime of references to it. You use `IUnknown::AddRef` to stabilize a copy of an interface pointer. It can also be called when the life of a

	cloned pointer must extend beyond the lifetime of the original pointer. The cloned pointer must be released by calling `IUnknown::Release`. Objects must be able to maintain (231)-1 outstanding pointer references. Therefore, the internal reference counter that `IUnknown::AddRef` maintains must be a 32-bit unsigned integer. Call this function for every new copy of an interface pointer that you make. For example, if you are passing a copy of a pointer back from a function, you must call `IUnknown::AddRef` on that pointer. You must also call `IUnknown::AddRef` on a pointer before passing it as an in-out parameter to a function; the function will call `IUnknown::Release` before copying the out-value on top of it.

Table 2.

Item	Description
Function	`IUnknown::Release`
Use	Decrements the reference count for the calling interface on a object. If the reference count on the object falls to 0, the object is freed from memory.
Prototype	`ULONG Release(void);`
Parameters	None.
Return value	Returns the resulting value of the reference count, which is used for diagnostic/testing purposes only.
Include file	MFC
Remark	If `IUnknown::AddRef` has been called on this object's interface n times and this is the **n+1th** call to `IUnknown::Release`, the implementation of `IUnknown::AddRef` must cause the interface pointer to free itself. When the released pointer is the only existing reference to an object (whether the object supports single or multiple interfaces), the implementation must free the object. Aggregation of objects restricts the ability to recover interface pointers. Call this function when you no longer need to use an interface pointer. If you are writing a function that takes an in-out parameter, call `IUnknown::Release` on the pointer you are passing in before copying the out-value on top of it.

Table 3.

----------End----------

Further reading and digging:

1. DCOM at MSDN.
2. COM+ at MSDN.
3. COM at MSDN.
4. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
5. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
6. MSDN Library
7. Windows data type.
8. Win32 programming Tutorial.
9. The best of C/C++, MFC, Windows and other related books.
10. Unicode and Multibyte character set: Story and program examples.