The Active Template Library (ATL) - Introduction

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 marshalling and intro to activeX control.

Index:

Intro

In this module, you'll take a look at the second framework (MFC being the first) now included within Microsoft Visual C++, the **Active Template Library** (ATL). You'll start by quickly revisiting the **Component Object Model** (COM) and looking at an alternative method of writing Module 23's CSpaceship object, illustrating that there is more than one way to write a COM class. (This will become important as you examine ATL's class composition methods.) Next you'll investigate the Active Template Library, focusing first on C++ templates and raw C++ smart pointers and how they might be useful in COM development. You'll cover the client side of ATL programming and examine some of ATL's smart pointers. Finally you'll check out the server side of ATL programming, re-implementing the Module 23 spaceship example using ATL to get a feel for ATL's architecture.

Revisiting the COM

The most important concept to understand about COM programming is that it is **interface-based**. As you saw in Module 23, you don't need real COM or even Microsoft runtime support to use interface-based programming. All you need is some discipline.
Think back to the spaceship example in Module 23. You started out with a single class named `CSpaceship` that implemented several functions. Seasoned C++ developers usually sit down at the computer and start typing a class like this:

```
class CSpaceship
{
    void Fly();
    int& GetPosition();
};
```

However, the procedure is a little different with interface-based development. Instead of writing the class directly, interface-based programming involves **spelling out an interface before implementing it**. In Module 23, the `Fly()` and `GetPosition()` functions were moved into an **abstract base class** named `IMotion`.

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

Then we inherited the `CSpaceship` class from the `IMotion` **interface** like this:

```
class CSpaceship : IMotion
{
    void Fly();
    int& GetPosition();
};
```

Notice that at this point the motion interface has been separated from its implementation. When practicing interface development, the **interface comes first**. You can work on the interface as you develop it, making sure it's complete while at the same time not over-bloated. But once the interface has been published (that is, once a lot of other developers have started coding to it), the interface is frozen and can never change.
This subtle distinction between **class-based programming** and **interface-based programming** seems to introduce some programming overhead. However, it turns out to be one of the key points to understanding COM. By collecting the `Fly()` and the `GetPosition()` functions in an interface, you've developed a binary signature. That is, by defining the interface ahead of time and talking to the class through the interface, **client** code has a potentially language-neutral way of talking to the class.
Gathering functions together into interfaces is itself quite powerful. Imagine you want to describe something other than a spaceship, an airplane, for example. It's certainly conceivable that an airplane would also have `Fly()` and `GetPosition()` functions. Interface programming provides a more advanced form of polymorphism, polymorphism at the interface level, not only at the single-function level.
Separating interface from implementation is the basis of interface-based development. The Component Object Model is centered on interface programming. COM enforces the distinction between **interface** and **implementation**. In COM, the only way client code can talk to an object is through an interface. However, gathering functions

together into interfaces isn't quite enough. There's one more ingredient needed, a mechanism for discovering functionality at runtime.

The Core Interface: IUnknown

The key element that makes COM different from ordinary interface programming is this rule: **the first three functions of every COM interface are the same**. The core interface in COM, `IUnknown`, looks like this:

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

Every COM interface derives from this interface (meaning the first three functions of every COM interface you ever see will be `QueryInterface()`, `AddRef()`, and `Release()`). To turn `IMotion` into a **COM interface**, derive it from `IUnknown` like this:

```
struct IMotion : IUnknown
{
    void Fly();
    int& GetPosition();
};
```

If you wanted these interfaces to work out-of-process, you'd have to make each function return an `HRESULT`. You'll see this when we cover **Interface Definition Language** (IDL) later in this module.
`AddRef()` and `Release()` deserve some mention because they are part of `IUnknown`. `AddRef()` and `Release()` allow an object to control its own lifetime if it chooses to. As a rule, clients are supposed to treat interface pointers like resources: clients acquire interfaces, use them, and release them when they are done using them. Objects learn about new references to themselves via `AddRef()`. Objects learn they have been unreferenced through the `Release()` function. Objects often use this information to control their lifetimes. For example, many objects self-destruct when their reference count reaches zero. Here's how some client code might use the spaceship:

```
void UseSpaceship()
{
    IMotion* pMotion = NULL;

    pMotion = GetASpaceship(); // This is a member of the
                               //  hypothetical Spaceship
                               //  API. It's presumably an
                               //  entry point into some DLL.
                               //  Returns an IMotion* and
                               //  causes an implicit AddRef.
    If(pMotion)
    {
        pMotion->Fly();
        int i = pMotion->GetPosition();
        pMotion->Release(); // done with this instance of CSpaceship
    }
}
```

The other (and more important) function within `IUnknown` is the first one: `QueryInterface()`. `QueryInterface()` is the COM mechanism for **discovering functionality at runtime**. If someone gives you a COM interface pointer to an object and you don't want to use that pointer, you can use the pointer to ask the object for a different interface to the same object. This mechanism, along with the fact that interfaces remain constant once published, are the key ingredients that allow COM-based software to evolve safely over time. The result is that you can add functionality to your COM software without breaking older versions of the clients running that software. In addition, clients have a widely recognized means of acquiring that new functionality once they know about it. For example, you add functionality to the implementation of `CSpaceship` by adding a new interface named

IVisual. Adding this interface makes sense because you can have objects in three-dimensional space that move in and out of view. You might also have an invisible object in three-dimensional space (a black hole, for example). Here's the IVisual interface:

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

A **client** might use the IVisual interface like this:

```
void UseSpaceship()
{
    IMotion* pMotion = NULL;

    pMotion = GetASpaceship(); // Implicit AddRef
    if(pMotion)
    {
        pMotion->Fly();
        int i = pMotion->GetPosition();

        IVisual* pVisual = NULL;
        PMotion->QueryInterface(IID_IVisual, (void**) &pVisual);
        // Implicit AddRef within QueryInterface

        if(pVisible)
        {
            pVisual->Display(); // uncloaking now
            pVisual->Release(); // done with this interface
        }
    }
    pMotion->Release(); // done with this instance of IMotion
}
```

Notice that the preceding code uses interface pointers very carefully: it uses them only if the interface was acquired properly, and then it releases the interface pointers when it is done using them. This is raw COM programming at the lowest level, you acquire an interface pointer, you use the interface pointer, and you release it when you're done with it.

Writing COM Code

As you can see, writing COM client code isn't a whole lot different from writing regular C++ code. However, the C++ classes that the client talks to are **abstract base classes**. Instead of calling operator new as you would in C++, you create COM objects and acquire COM interfaces by explicitly calling some sort of API function. And instead of deleting the object outright, you simply follow the COM interface rule of balancing calls to AddRef() with calls to Release().
What does it take to get the COM class up and running? You saw how to do it using MFC in Module 23. Here's another example of implementing CSpaceship as a COM class. This example uses the multiple inheritance approach to writing COM classes. That is, the C++ class inherits from several interfaces and then implements the union of all the functions (including IUnknown, of course).

```
struct CSpaceship : IMotion, IDisplay
{
    ULONG m_cRef;
    int m_nPosition;

    CSpaceship() : m_cRef(0), m_nPosition(0)
    { }

    HRESULT QueryInterface(REFIID riid, void** ppv);
```

```
            ULONG AddRef()
            { return InterlockedIncrement(&m_cRef); }

            ULONG Release()
            {
                ULONG cRef = InterlockedIncrement(&m_cRef);
                if(cRef == 0)
                {
                    delete this;
                    return 0;
                }
                else
                    return m_cRef;
            }

            // IMotion functions:
            void Fly()
            {
                // Do whatever it takes to fly here
            }

           int GetPosition()
           {
                return m_nPosition;
           }

            // IVisual functions:
            void Display()
            {
                // Uncloak
            }
        };
```

COM Classes Using Multiple Inheritance

If you're used to seeing plain C++ code, the preceding code might look a little strange to you. This is a less common form of multiple inheritance called **interface inheritance**. Most C++ developers are used to an implementation inheritance in which the derived class inherits everything from the base class, including the implementation. Interface inheritance simply means the derived class inherits the interfaces of the base class. The preceding code effectively adds two data members to the `CSpaceship` class, a `vptr` for each implied vtable.
When using the multiple inheritance approach to implementing interfaces, each interface shares `CSpaceship`'s implementation of `IUnknown`. This sharing illustrates a rather esoteric yet important concept known as **COM identity**. The basic idea of COM identity is that `IUnknown` is the `void*` of COM. `IUknown` is the one interface guaranteed to be hanging off any object, and you can always get to it. COM identity also says (in the previous example) the client can call `QueryInterface()` through the `CSpaceship` `IMotion` interface to get the `IVisible` interface. Conversely, the client can call `QueryInterface()` through the `CSpaceship` `IVisible` interface to get the `IMotion` interface. Finally, the client can call `QueryInterface()` through `IUnknown` to acquire the `IMotion` or the `IVisible` interface, and the client can call `QueryInterface()` through either `IMotion` or `IVisual` to get a pointer to `IUnknown`. To learn more about COM identity, see Essential COM by Don Box (Addison-Wesley, 1997) or Inside COM by Dale Rogerson (Microsoft Press, 1997).
Often you'll see **COM classes** illustrated with "lollipop" diagrams depicting the **interfaces implemented by a COM class**.
The multiple inheritance method of implementing `CSpaceship` automatically fulfills the rules of COM identity. Note that all calls to `QueryInterface()`, `AddRef()`, and `Release()` land in the same place in the C++ class, regardless of the interface through which they were called.
This is more or less the essence of COM. As a COM developer, your job is to **create useful services and expose them through COM interfaces**. At the most basic level, this means wiring up some function tables to follow COM's identity rules. You've seen two ways to accomplish this so far. Module 23 showed you how to do it using **nested classes** and **MFC**. This module just showed you how to write a COM class using multiple inheritance in

C++. However, in addition to interface programming and writing classes to implement interfaces, there are several other pieces to the COM puzzle.

The COM Infrastructure

Once you get your mind around the concept of interface-based programming, quite a few details need implementation in order to get the class to mix in with the rest of the system. These details often overshadow the fundamental beauty of COM.
To start off with, COM classes **need a place to live**, so you must package them in either an **EXE** or a **DLL**. In addition, each COM class you write needs an **accompanying class object** (often referred to as a **class factory**). The way in which a COM server's class object is exposed differs depending upon how you package the COM class (in a DLL or an EXE). The server lifetime also needs to be considered. The server should stay in memory for as long as it's needed, and it should go away when it's not needed. To accomplish this, servers maintain global lock counts indicating the number of objects with extant interface pointers. Finally, well-behaved servers insert the necessary values in the Windows Registry so that client software can easily activate them.
You've spent a lot of time looking at MFC while reading this book. As you saw in Module 23, MFC takes care of most of the COM-based details for you. For example, `CCmdTarget` has an implementation of `IUnknown`. MFC has even created **C++ classes** and **macros** to implement class objects (such as `COleObjectFactory`, `COleTemplateServer`, `DECLARE_OLE_CREATE`, and `IMPLEMENT_OLE_CREATE`) that will put most of the correct entries into the Registry. MFC has the easiest-to-implement, zippiest version of IDispatch around—all you need is a `CCmdTarget` object and ClassWizard. If you decide OLE Documents or ActiveX Documents are your thing, MFC provides standard implementations of the Object Linking and Embedding and ActiveX Document protocols. Finally, MFC remains hands-down the easiest way to write fast, powerful **ActiveX controls**. You can write ActiveX controls in Microsoft Visual Basic, but you don't have quite as much flexibility. These are all great features. However, using MFC has a downside.
To get these features, you need to buy into MFC 100%. Now, that's not necessarily a bad idea. However, you should be aware of the cost of entry when you decide to use MFC. MFC is big. It has to be, it's a C++ framework with many capabilities.

A New Framework

As you can see from the examples we've looked at so far, implementing COM classes and making them available to clients involves writing a great deal of code, code that remains the same from one class implementation to another. `IUnknown` implementations are generally the same for every COM class you encounter, the main difference between them is the **interfaces exposed by each class**.
But just as you no longer need to understand assembly language to get software working these days, pretty soon you'll no longer need to understand all the nuances of `IUnknown` and COM's relationship to C++ to get your COM-based software up and running. You're not quite at that stage, but the Active Template Library (ATL) from Microsoft is a great first step in that direction. However, ATL does not absolve you from learning the important concepts behind COM, such as **apartments** and **remoting**. Before diving into ATL, let's take a quick peek at where COM and ATL fit into the big picture.

ActiveX, OLE, and COM

COM is simply the plumbing for a series of **higher-level application integration technologies** consisting of such items as ActiveX Controls and OLE Documents. These technologies define protocols based on COM interfaces. For example, for a COM object to qualify as a minimal OLE Document object, that COM object has to implement at least three interfaces, `IPersistStorage`, `IOleObject`, and `IDataObject`. You might choose to implement the higher-level features of OLE Documents and controls. However, it makes more sense to let some sort of application framework do the grunt work. Of course, that's why there's MFC. For more information about how to implement higher-level features in raw C++, see Kraig Brockschmidt's Inside OLE, 2d. ed. (Microsoft Press, 1995).

ActiveX, MFC, and COM

While the pure plumbing of COM is quite interesting by itself (it's simply amazing to see how COM remoting works), the higher-level features are what sell applications. MFC is a huge framework geared toward creating entire Windows applications. Inside MFC, you'll find tons of utility classes, a data management/rendering mechanism (the Document-View architecture), as well as support for OLE Documents, drag and drop, Automation, and ActiveX Controls. You probably don't want to develop an OLE Document application from scratch; you're much better off

using MFC. However, if you need to create **a small or medium-size COM-based service**, you might want to turn away from MFC so you don't have to include all the baggage MFC maintains for the higher-level features.
You can use raw C++ to create COM components, but doing so forces you to spend a good portion of your time hacking out the boilerplate code (`IUnknown` and class objects, for example). Using MFC to write COM-based applications turns out to be a less painful way of adding the big-ticket items to your application, but it's difficult to write lightweight COM classes in MFC. **ATL sits between pure C++ and MFC as a way to implement COM-based software without having to type in the boilerplate code or buy into all of MFC's architecture**. ATL is basically **a set of** C++ templates and other kinds of support for writing COM classes.

The ATL Roadmap

If you look at the source code for ATL, you'll find ATL consists of a collection of header files and C++ source code files. Most of it resides inside **ATL's Include directory**. Here's a rundown of some of the ATL files and what's inside each of them.

ATLBASE.H

This file contains:

- ATL's function typedefs.
- Structure and macro definitions.
- Smart pointers for managing COM interface pointers.
- Thread synchronization support classes.
- Definitions for `CComBSTR`, `CComVariant`, threading, and apartment support.

ATLCOM.H

This file contains:

- Template classes for class object/class factory support.
- IUnknown implementations.
- Support for tear-off interfaces.
- Type information management and support.
- ATL's IDispatch implementation.
- COM enumerator templates.
- Connection point support.

ATLCONV.CPP and ATLCONV.H

These two source code files include support for Unicode conversions.

ATLCTL.CPP and ATLCTL.H

These two files contain:

- The source code for ATL's IDispatch client support and event firing support.
- `CComControlBase`.
- The OLE embedding protocol support for controls.
- Property page support.

ATLIFACE.IDL and ATLIFACE.H

`ATLIFACE.IDL` (which generates `ATLIFACE.H`) includes an ATL-specific interface named `IRegistrar`.

ATLIMPL.CPP

`ATLIMPL.CPP` implements such classes as `CComBSTR`, which is declared in `ATLBASE.H`.

ATLWIN.CPP and ATLWIN.H

These files provide windowing and user-interface support, including:

- A message-mapping mechanism.
- A windowing class.
- Dialog support.

STATREG.CPP and STATREG.H

ATL features a COM component named the Registrar that handles putting appropriate entries into the Registry. The code for implementing this feature is in STATREG.H and STATREG.CPP.
Let's start our excursions into ATL by examining ATL's support for client-side COM development.

Client-Side ATL Programming

There are basically two sides to ATL, **client-side support** and **object-side support**. By far the largest portion of support is on the object side because of all the code necessary to implement ActiveX controls. However, the client-side support provided by ATL turns out to be useful and interesting also. Let's take a look at the client side of ATL. Because C++ templates are the cornerstone of ATL, we'll take a little detour first to examine them.

C++ Templates

The key to understanding the Active Template Library is understanding C++ templates. Despite the intimidating template syntax, the concept of templates is fairly straightforward. C++ templates are sometimes called compiler-approved macros, which is an appropriate description. Think about what macros do: when the preprocessor encounters a macro, the preprocessor looks at the macro and expands it into regular C++ code. But the problem with macros is that they are sometimes error-prone and they are **never type-safe**. If you use a macro and pass an incorrect parameter, the compiler won't complain but your program might very well crash. Templates, however, are like type-safe macros. When the compiler encounters a template, the compiler expands the template just as it would a macro. But because templates are type-safe, the compiler catches any type problems before the user encounters them.
Using templates to reuse code is different from what you're used to with conventional C++ development. Components written using templates reuse code by template substitution rather than by inheriting functionality from base classes. All the boilerplate code from templates is literally pasted into the project.
The archetypal example of using a template is a **dynamic array**. Imagine you need an array for holding integers. Rather than declaring the array with a fixed size, you want the array to grow as necessary. So you develop the array as a C++ class. Then someone you work with gets wind of your new class and says that he or she needs the exact same functionality. However, this person wants to use floating point numbers in the array. Rather than pumping out the exact same code (except for using a different type of data), you can use a C++ template.
Here's an example of how you might use templates to solve the problem described above. The following is a dynamic array implemented as a template:

```
template <class T>
class DynArray
{
public:
    DynArray();
    ~DynArray(); // clean up and do memory management
    int Add(T Element); // adds an element and does
                        //  memory management
    void Remove(int nIndex); // remove element and
                             //  do memory management
    T GetAt(int nIndex) const;
    int GetSize();
private:
    T* TArray;
    int m_nArraysize;
};

void UseDynArray()
{
    DynArray<int> intArray;
    DynArray<float> floatArray;

    intArray.Add(4);
    floatArray.Add(5.0);

    intArray.Remove(0);
    floatArray.Remove(0);

    int x = intArray.GetAt(0);
    float f = floatArray.GetAt(0);
}
```

As you can imagine, creating templates is useful for implementing boilerplate COM code, and templates are the mechanism ATL uses for providing COM support. The previous example is just one of the many uses available for templates. Not only are templates useful for applying type information to a certain kind of data structure, they're also useful for encapsulating algorithms. You'll see how when you take a closer look at ATL. Let's take a look at the Active Template Library to see what comes with it.

Smart Pointers

One of the most common uses of templates is for **smart pointers**. The traditional C++ literature calls C++'s built-in pointers "dumb" pointers. That's not a very nice name, but normal C++ pointers don't do much **except point**. It's often up to the client to perform details such as pointer initialization.
As an example, let's model two types of software developer using C++ classes. You can start by creating the classes: `CVBDeveloper` and `CCPPDeveloper`.

```
class CVBDeveloper
{
public:
    CVBDeveloper()
    { }
    ~CVBDeveloper()
    { AfxMessageBox("I used VB, so I got home early."); }
    virtual void DoTheWork()
    { AfxMessageBox("Write them forms"); }
};

class CCPPDeveloper
{
public:
    CCPPDeveloper()
    {   }
    ~CCPPDeveloper()
    { AfxMessageBox("Stay at work and fix those pointer problems"); }
    virtual void DoTheWork()
    { AfxMessageBox("Hacking C++ code"); }
};
```

The Visual Basic developer and the C++ developer both have functions for eliciting optimal performance. Now imagine some client code that looks like this:

```
//UseDevelopers.CPP

void UseDevelopers()
{
    CVBDeveloper* pVBDeveloper;
    ...
    ...
    ...
    // The VB Developer pointer needs
```

```
        //  to be initialized
        //  sometime. But what if
        //  you forget to initialize and later
        //  on do something like this:
        if(pVBDeveloper)
        {
            // Get ready for fireworks
            //  because pVBDeveloper is
            //  NOT NULL, it points
            //  to some random data.
            c->DoTheWork();
        }
    }
```

In this case, the client code forgot to initialize the `pVBDeveloper` pointer to `NULL`. (Of course, this never happens in real life!) Because `pVBDeveloper` contains a non-NULL value (the value is actually whatever happened to be on the stack at the time), the test to make sure the pointer is valid succeeds when in fact you're expecting it to fail. The client gleefully proceeds, believing all is well. The client crashes, of course, because the client is "calling into darkness." (Who knows where `pVBDeveloper` is pointing, probably to nothing that even resembles a Visual Basic developer.) Naturally, you'd like some mechanism for ensuring that the pointers are initialized. This is where smart pointers come in handy.

Now imagine a second scenario. Perhaps you'd like to plug a little extra code into your developer-type classes that performs some sort of operation common to all developers. For example, perhaps you'd like all the developers to do some design work before they begin coding. Consider the earlier VB developer and C++ developer examples. When the client calls `DoTheWork()`, the developer gets right to coding without proper design, and he or she probably leaves the poor clients in a lurch. What you'd like to do is add a very generic hook to the developer classes so they make sure the design is done before beginning to code.

The C++ solution to coping with these problems is called a smart pointer. Let's find out exactly what a smart pointer is.

Giving C++ Pointers Some Brains

Remember that a smart pointer is a **C++ class for wrapping pointers**. By wrapping a pointer in a class (and specifically, a template), you can make sure certain operations are taken care of automatically instead of deferring mundane, boilerplate-type operations to the client. One good example of such an operation is to make sure pointers are initialized correctly so that embarrassing crashes due to randomly assigned pointers don't occur. Another good example is to make certain that boilerplate code is executed before function calls are made through a pointer. Let's invent a smart pointer for the developer model described earlier. Consider a template-based class named `SmartDeveloper`:

```
    template<class T>
    class SmartDeveloper
    {
        T* m_pDeveloper;

    public:
        SmartDeveloper(T* pDeveloper)
        {
            ASSERT(pDeveloper != NULL);
            m_pDeveloper = pDeveloper;
        }
        ~SmartDeveloper()
        { AfxMessageBox("I'm smart so I'll get paid."); }
        SmartDeveloper &operator=(const SmartDeveloper& rDeveloper)
        { return *this; }
        T* operator->() const
        {
            AfxMessageBox("About to de-reference pointer. Make /
                            sure everything's okay. ");
            return m_pDeveloper;
        }
    };
```

The `SmartDeveloper` template listed above wraps a pointer, any pointer. Because the `SmartDeveloper` class is based on a template, it can provide generic functionality regardless of the type associated with the class. Think of templates as compiler-approved macros: declarations of classes (or functions) whose code can **apply to any type of data**.
We want the smart pointer to handle all developers, including those using VB, Visual C++, Java, and Delphi (among others). The template `<class T>` statement at the top accomplishes this. The `SmartDeveloper` template includes a pointer (`m_pDeveloper`) to the type of developer for which the class will be defined. The `SmartDeveloper` constructor takes a pointer to that type as a parameter and assigns it to `m_pDeveloper`. Notice that the constructor generates an assertion if the client passes a NULL parameter to construct `SmartDeveloper`.
In addition to wrapping a pointer, the `SmartDeveloper` implements several operators. The most important one is the "`->`" operator (the member selection operator). This operator is the workhorse of any smart pointer class. Overloading the member selection operator is what turns a regular class into a smart pointer. Normally, using the member selection operator on a regular C++ dumb pointer tells the compiler to select a member belonging to the class or structure being pointed to. By overriding the member selection operator, you provide a way for the client to hook in and call some boilerplate code every time that client calls a method. In the SmartDeveloper example, the smart developer makes sure the work area is in order before working. This example is somewhat contrived. In real life, you might want to put in a debugging hook, for example.
Adding the `->` operator to the class causes the class to behave like C++'s built-in pointer. To behave like native C++ pointers in other ways, smart pointer classes need to implement the other standard operators such as the de-referencing and assignment operators.

Using Smart Pointers

Using smart pointers is really no different from using the regular built-in C++ pointers. Let's start by looking at a client that uses plain vanilla developer classes:

```
    void UseDevelopers()
    {
        CVBDeveloper VBDeveloper;
        CCPPDeveloper CPPDeveloper;

        VBDeveloper.DoTheWork();
        CPPDeveloper.DoTheWork();
    }
```

No surprises here, executing this code causes the developers simply to come in and do the work. However, you want to use the smart developers, the ones that make sure the design is done before actually starting to hack. Here's the code that wraps the VB developer and C++ developer objects in the smart pointer class:

```
    void UseSmartDevelopers
    {
        CVBDeveloper VBDeveloper;
        CCPPDeveloper CPPDeveloper;

        SmartDeveloper<CVBDeveloper> smartVBDeveloper(&VBDeveloper);
        SmartDeveloper<CCPPDeveloper> smartCPPDeveloper(&CPPDeveloper);

        smartVBDeveloper->DoTheWork();
        smartCPPDeveloper->DoTheWork();
    }
```

Instead of bringing in any old developer to do the work (as in the previous example), the client asks the smart developers to do the work. The smart developers will automatically prepare the design before proceeding with coding.

Smart Pointers and COM

While the last example was fabricated to make an interesting story, smart pointers do have useful applications in the real world. One of those applications is to make **client-side COM programming easier**.
Smart pointers are frequently used to implement reference counting. Because reference counting is a very generic operation, hoisting client-side reference count management up into a smart pointer makes sense.
Because you're now familiar with the Microsoft Component Object Model, you understand that **COM objects expose interfaces**. To C++ clients, interfaces are simply pure abstract base classes, and C++ clients treat interfaces more or less like normal C++ objects. However, as you discovered in previous modules, COM objects are a bit different from regular C++ objects. **COM objects live at the binary level**. As such, they are created and destroyed using language- independent means. COM objects are created via **API functions calls**. Most COM objects use a reference count to know when to delete themselves from memory. Once a COM object is created, a client object can refer to it in a number of ways by referencing multiple interfaces belonging to the same COM object. In addition, several different clients can talk to a single COM object. In these situations, the COM object must stay alive for as long as it is referred to. Most COM objects destroy themselves when they're no longer referred to by any clients. COM objects use reference counting to accomplish this self-destruction.
To support this reference-counting scheme, COM defines a couple of rules for managing COM interfaces from the client side. The first rule is that creating a new copy of a COM interface should result in bumping the object's reference count up by one. The second rule is that clients should release interface pointers when they have finished with them. Reference counting is one of the more difficult aspects of COM to get right, especially from the client side. Keeping track of COM interface reference counting is a perfect use of smart pointers.
For example, the smart pointer's constructor might take the live interface pointer as an argument and set an internal pointer to the live interface pointer. Then the destructor might call the interface pointer's Release function to release the interface so that the interface pointer will be released automatically when the smart pointer is deleted or falls out of scope. In addition, the smart pointer can help manage COM interfaces that are copied.
For example, imagine you've created a COM object and you're holding on to the interface. Suppose you need to make a copy of the interface pointer (perhaps to pass it as an out parameter). At the native COM level, you'd perform several steps. First you must release the old interface pointer. Next you need to copy the old pointer to the new pointer. Finally you must call `AddRef()` on the new copy of the interface pointer. These steps need to occur regardless of the interface being used, making this process ideal for boilerplate code. To implement this process in the smart pointer class, all you need to do is override the assignment operator. The client can then assign the old pointer to the new pointer. The smart pointer does all the work of managing the interface pointer, relieving the client of the burden.

ATL's Smart Pointers

Much of ATL's support for client-side COM development resides in a pair of ATL smart pointers: `CComPtr` and `CComQIPtr`. `CComPtr` is a basic smart pointer that wraps COM interface pointers. `CComQIPtr` adds a little more smarts by associating a GUID (for use as the interface ID) with a smart pointer. Let's start by looking at `CComPtr`.

CComPtr

Here's an abbreviated version of `CComPtr` showing its most important parts:

```
template <class T>
class CComPtr
{
public:
    typedef T _PtrClass;
    CComPtr() {p=NULL;}
    CComPtr(T* lp)
    {
        if ((p = lp) != NULL) p->AddRef();
    }
    CComPtr(const CComPtr<T>& lp)
    {
        if ((p = lp.p) != NULL) p->AddRef();
    }
    ~CComPtr() {if (p) p->Release();}
    void Release() {if (p) p->Release(); p=NULL;}
    operator T*() {return (T*)p;}
```

```
        T& operator*() {_ASSERTE(p!=NULL); return *p; }
        T** operator&() { _ASSERTE(p==NULL); return &p; }
        T* operator->() { _ASSERTE(p!=NULL); return p; }
        T* operator=(T* lp)
        {return (T*)AtlComPtrAssign((IUnknown**)&p, lp);}
        T* operator=(const CComPtr<T>& lp)
        {
            return (T*)AtlComPtrAssign((IUnknown**)&p, lp.p);
        }
        T* p;
    };
```

`CComPtr` is a fairly basic smart pointer. Notice the data member `p` of type `T` (the type introduced by the template parameter). `CComPtr`'s constructor performs an `AddRef()` on the pointer while the destructor releases the pointer, no surprises here. `CComPtr` also has all the necessary operators for wrapping a COM interface. Only the assignment operator deserves special mention. The assignment does a raw pointer re-assignment. The assignment operator calls a function named `AtlComPtrAssign()`:

```
    ATLAPI_(IUnknown*) AtlComPtrAssign(IUnknown** pp, IUnknown* lp)
    {
        if (lp != NULL)
            lp->AddRef();
        if (*pp)
            (*pp)->Release();
        *pp = lp;
        return lp;
    }
```

`AtlComPtrAssign()` does a blind pointer assignment, `AddRef()`-ing the assignee before calling `Release()` on the assignor. You'll soon see a version of this function that calls `QueryInterface()`. `CComPtr`'s main strength is that it helps you manage the reference count on a pointer to some degree.

Using `CComPtr`

In addition to helping you manage `AddRef()` and `Release()` operations, `CComPtr` can help you manage code layout. Looking at a bit of code will help illustrate the usefulness of `CComPtr`. Imagine that your client code needs three interface pointers to get the work done as shown here:

```
    void GetLottaPointers(LPUNKNOWN pUnk)
    {
        HRESULT hr;
        LPPERSIST pPersist;
        LPDISPATCH pDispatch;
        LPDATAOBJECT pDataObject;
        hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&pPersist);
        if(SUCCEEDED(hr))
        {
            hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *)&pDispatch);
            if(SUCCEEDED(hr))
            {
                hr = pUnk->QueryInterface(IID_IDataObject, (LPVOID *)
    &pDataObject);
                if(SUCCEEDED(hr))
                {
                    DoIt(pPersist, pDispatch, pDataObject);
                    pDataObject->Release();
                }
                pDispatch->Release();
            }
            pPersist->Release();
        }
    }
```

You could use the controversial `goto` statement (and risk facing derisive comments from your co-workers) to try to make your code look cleaner, like this:

```
    void GetLottaPointers(LPUNKNOWN pUnk)
    {
        HRESULT hr;
        LPPERSIST pPersist; LPDISPATCH pDispatch;
        LPDATAOBJECT pDataObject;

        hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&pPersist);
        if(FAILED(hr)) goto cleanup;

        hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *) &pDispatch);
        if(FAILED(hr)) goto cleanup;

        hr = pUnk->QueryInterface(IID_IDataObject, (LPVOID *)
    &pDataObject);
        if(FAILED(hr)) goto cleanup;

        DoIt(pPersist, pDispatch, pDataObject);

    cleanup:
        if (pDataObject) pDataObject->Release();
        if (pDispatch) pDispatch->Release();
        if (pPersist) pPersist->Release();
    }
```

That may not be as elegant a solution as you would like. Using `CComPtr` makes the same code a lot prettier and much easier to read, as shown here:

```
    void GetLottaPointers(LPUNKNOWN pUnk)
    {
        HRESULT hr;
        CComPtr<IUnknown> persist;
        CComPtr<IUnknown> dispatch;
        CComPtr<IUnknown> dataobject;

        hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&persist);
        if(FAILED(hr)) return;

        hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *) &dispatch);
        if(FAILED(hr)) return;

        hr = pUnk->QueryInterface(IID_IDataObject, (LPVOID *) &dataobject);
        if(FAILED(hr)) return;

        DoIt(pPersist, pDispatch, pDataObject);

        // Destructors call release...
    }
```

At this point, you're probably wondering why `CComPtr` doesn't wrap `QueryInterface()`. After all, `QueryInterface()` is a hot spot for reference counting. Adding `QueryInterface()` support for the smart pointer requires some way of associating a GUID with the smart pointer. `CComPtr` was introduced in the first version of ATL. Rather than disrupt any existing code base, Microsoft introduced a beefed-up version of `CComPtr` named `CComQIPtr`.

CComQIPtr

Here's part of CComQIPtr's definition:

```
template <class T, const IID* piid = &__uuidof(T)>
class CComQIPtr
{
public:
    typedef T _PtrClass;
    CComQIPtr() {p=NULL;}
    CComQIPtr(T* lp)
    {
        if ((p = lp) != NULL)
            p->AddRef();
    }
    CComQIPtr(const CComQIPtr<T,piid>& lp)
    {
        if ((p = lp.p) != NULL)
            p->AddRef();
    }
    CComQIPtr(IUnknown* lp)
    {
        p=NULL;
        if (lp != NULL)
            lp->QueryInterface(*piid, (void **)&p);
    }
    ~CComQIPtr() {if (p) p->Release();}
    void Release() {if (p) p->Release(); p=NULL;}
    operator T*() {return p;}
    T& operator*() {_ASSERTE(p!=NULL); return *p; }
    T** operator&() { _ASSERTE(p==NULL); return &p; }
    T* operator->() {_ASSERTE(p!=NULL); return p; }
    T* operator=(T* lp)
    {
        return (T*)AtlComPtrAssign((IUnknown**)&p, lp);
    }
    T* operator=(const CComQIPtr<T,piid>& lp)
    {
        return (T*)AtlComPtrAssign((IUnknown**)&p, lp.p);
    }
    T* operator=(IUnknown* lp)
    {
        return (T*)AtlComQIPtrAssign((IUnknown**)&p, lp, *piid);
    }
    bool operator!(){return (p == NULL);}
    T* p;
};
```

What makes CComQIPtr different from CComPtr is the second template parameter, **piid**, the interfaces's GUID. This smart pointer has several constructors: a default constructor, a copy constructor, a constructor that takes a raw interface pointer of unspecified type, and a constructor that accepts an IUnknown interface as a parameter. Notice in this last constructor that if the developer creates an object of this type and initializes it with a plain old IUnknown pointer, CComQIPtr calls QueryInterface() using the GUID template parameter. Also notice that the assignment to an IUnknown pointer calls AtlComQIPtrAssign() to make the assignment. As you can imagine, AtlComQIPtrAssign() performs a QueryInterface() under the hood using the GUID template parameter.

Using CComQIPtr

Here's how you might use CComQIPtr in some COM client code:

```
void GetLottaPointers(ISomeInterface* pSomeInterface)
{
    HRESULT hr;
    CComQIPtr<IPersist, &IID_IPersist> persist;
    CComQIPtr<IDispatch, &IID_IDispatch> dispatch;
    CComPtr<IDataObject, &IID_IDataObject> dataobject;

    dispatch = pSomeInterface;   // implicit QI
    persist = pSomeInterface;    //  implicit QI
    dataobject = pSomeInterface; //  implicit QI

    DoIt(persist, dispatch, dataobject); // send to a function
                                         // that needs IPersist*,
                                         // IDispatch*, and
                                         // IDataObject*

    // Destructors call release...
}
```

The `CComQIPtr` is useful whenever you want the Java-style or Visual Basic-style type conversions. Notice that the code listed above didn't require any calls to `QueryInterface()` or `Release()`. Those calls happened automatically.

ATL Smart Pointer Problems

Smart pointers can be quite convenient in some places (as in the `CComPtr` example where we eliminated the `goto` statement). Unfortunately, C++ smart pointers aren't the panacea that programmers pray for to solve their reference-counting and pointer-management problems. Smart pointers simply move these problems to a different level.
One situation in which to be very careful with smart pointers is when converting from code that is not smart-pointer based to code that uses the ATL smart pointers. The problem is that the ATL smart pointers don't hide the `AddRef()` and `Release()` calls. This just means you need to take care to understand how the smart pointer works rather than be careful about how you call `AddRef()` and `Release()`. For example, imagine taking this code:

```
void UseAnInterface()
{
    IDispatch* pDispatch = NULL;

    HRESULT hr = GetTheObject(&pDispatch);
    if(SUCCEEDED(hr))
    {
        DWORD dwTICount;
        pDispatch->GetTypeInfoCount(&dwTICount);
        pDispatch->Release();
    }
}
```

and capriciously converting the code to use a smart pointer like this:

```
void UseAnInterface()
{
    CComPtr<IDispatch> dispatch = NULL;

    HRESULT hr = GetTheObject(&dispatch);
    if(SUCCEEDED(hr))
    {
        DWORD dwTICount;
        dispatch->GetTypeInfoCount(&dwTICount);
        dispatch->Release();
    }
}
```

Because `CComPtr` and `CComQIPtr` do not hide calls to `AddRef()` and `Release()`, this blind conversion causes a problem when the release is called through the dispatch smart pointer. The `IDispatch` interface performs its own release, so the code above calls `Release()` twice, the first time explicitly through the call `dispatch->Release()` and the second time implicitly at the function's closing curly bracket.
In addition, ATL's smart pointers include the implicit cast operator that allows smart pointers to be assigned to raw pointers. In this case, what's actually happening with the reference count starts to get confusing.
The bottom line is that while smart pointers make some aspect of client-side COM development more convenient, they're not foolproof. You still have to have some degree of knowledge about how smart pointers work if you want to use them safely.

Server-Side ATL Programming

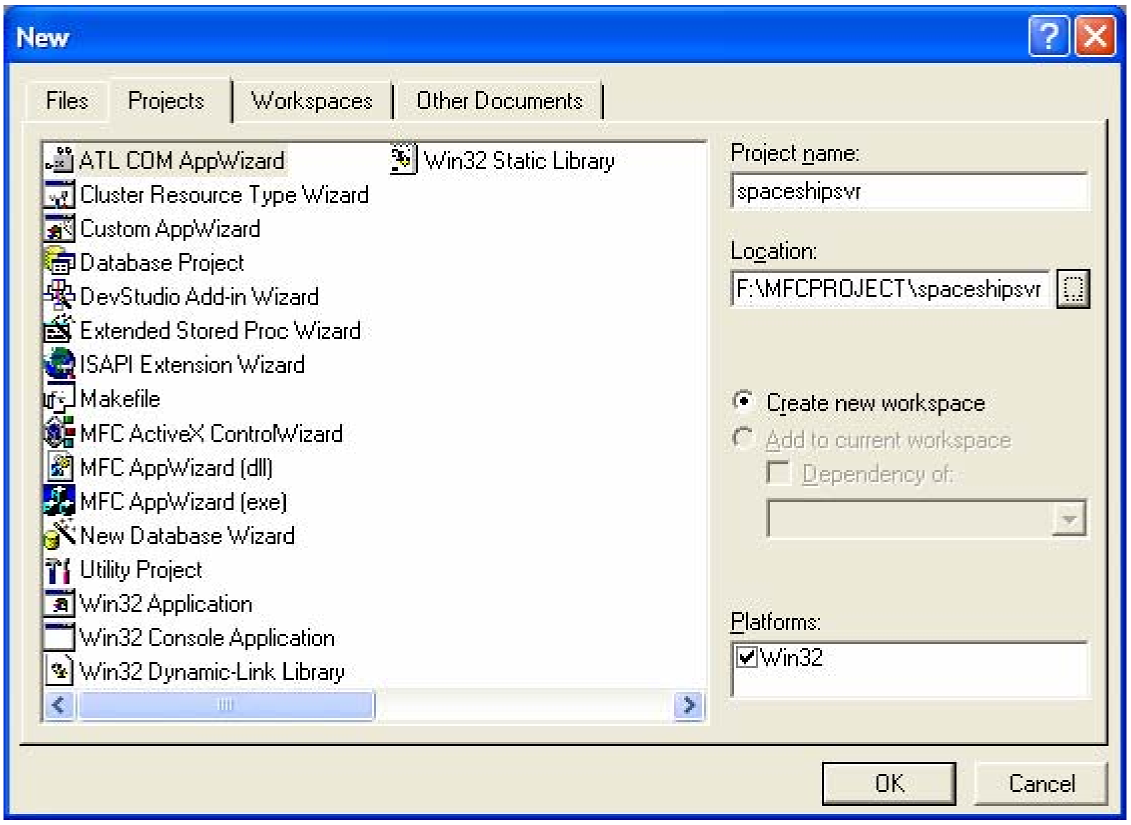
We've covered ATL's client-side support. While a fair amount of ATL is devoted to client-side development aids (such as smart pointers and `BSTR` wrappers), the bulk of ATL exists to support COM-based servers, which we'll cover next. First you'll get an overview of ATL in order to understand how the pieces fit together. Then you'll re-implement the spaceship example in ATL to investigate ATL's Object Wizard and get a good feel for what it takes to write COM classes using ATL.

ATL and COM Classes

Your job as a COM class developer is to wire up the function tables to their implementations and to make sure `QueryInterface()`, `AddRef()`, and `Release()` work as advertised. How you get that to happen is your own business. As far as users are concerned, they couldn't care less what methods you use. You've seen two basic approaches so far, the raw C++ method using multiple inheritance of interfaces and the MFC approach using macros and nested classes. The ATL approach to implementing COM classes is somewhat different from either of these approaches.
Compare the raw C++ approach to MFC's approach. Remember that one way of developing COM classes using raw C++ involves multiply inheriting a single C++ class from at least one COM interface and then writing all the code for the C++ class. At that point, you've got to add any extra features (such as supporting `IDispatch` or COM aggregation) by hand. The MFC approach to COM classes involves using macros that define nested classes (with one nested class implementing each interface). MFC supports `IDispatch` and COM aggregation, you don't have to do a lot to get those features up and running. However, it's very difficult to paste any new interfaces onto a COM class without a lot of typing. As you saw in Module 23, MFC's COM support uses some lengthy macros.
The ATL approach to composing COM classes requires inheriting a C++ class from several template-based classes. However, Microsoft has already done the work of implementing `IUnknown` for you through the class templates within ATL.

The Spaceshipsvr From Scratch

Before we dig the story further, let build **Spaceshipsvr** program, a COM class from scratch. As usual, in Visual C++, select **File New**.



New

Files | Projects | Workspaces | Other Documents

ATL COM AppWizard
Cluster Resource Type Wizard
Custom AppWizard
Database Project
DevStudio Add-in Wizard
Extended Stored Proc Wizard
ISAPI Extension Wizard
Makefile
MFC ActiveX ControlWizard
MFC AppWizard (dll)
MFC AppWizard (exe)
New Database Wizard
Utility Project
Win32 Application
Win32 Console Application
Win32 Dynamic-Link Library
Win32 Static Library

Project name: spaceshipsvr

Location: F:\MFCPROJECT\spaceshipsvr

[✓ Create new workspace] Add to current workspace

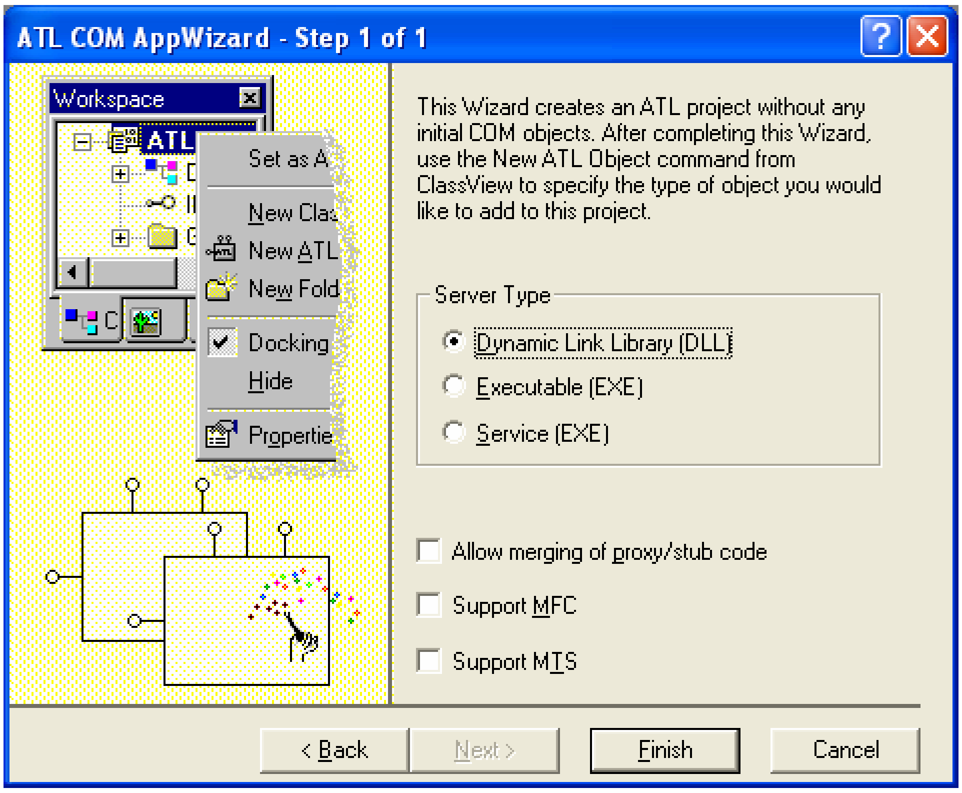
Dependency of:

Platforms: [✓ Win32]

OK | Cancel

Figure 1: **Spaceshipsvr** – ATL COM AppWizard new project dialog.

Select **Dynamic Link Library (DLL)** for the server type. Leave other options to default. Click **Finish** button.



ATL COM AppWizard - Step 1 of 1

This Wizard creates an ATL project without any initial COM objects. After completing this Wizard, use the New ATL Object command from ClassView to specify the type of object you would like to add to this project.

Server Type

[✓ Dynamic Link Library (DLL)] Executable (EXE) Service (EXE)

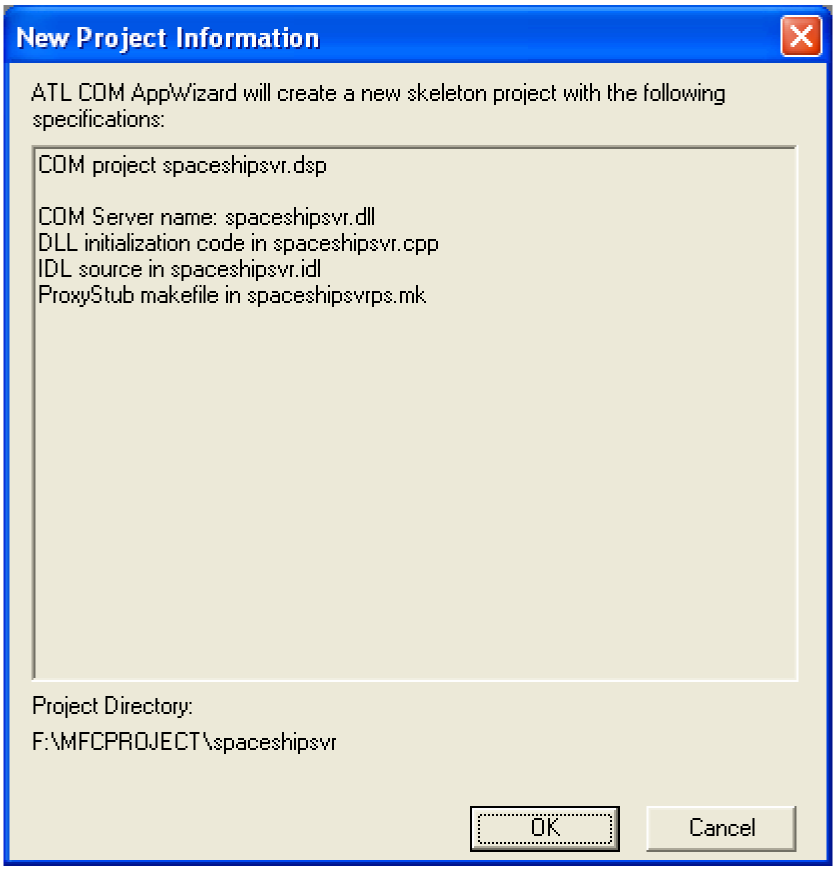
Allow merging of proxy/stub code: unchecked

Support MFC: unchecked

Support MTS: unchecked

< Back | Next > | Finish | Cancel

Figure 2: ATL COM AppWizard step 1 of 1.


New Project Information

ATL COM AppWizard will create a new skeleton project with the following specifications:

COM project spaceshipsvr.dsp

COM Server name: spaceshipsvr.dll
DLL initialization code in spaceshipsvr.cpp
IDL source in spaceshipsvr.idl
ProxyStub makefile in spaceshipsvrps.mk

Project Directory:
F:\MFCPROJECT\spaceshipsvr

Figure 3: **Spaceshipsvr** project summary.

Add new ATL object. Select **Insert New ATL Object**.

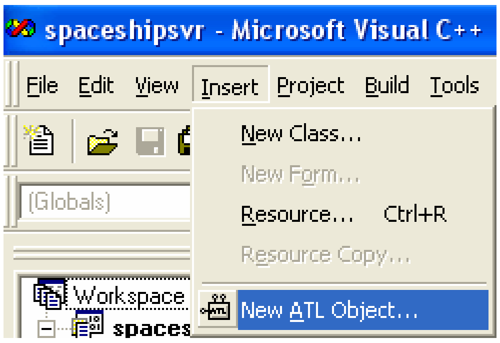


Figure 4: Adding new ATL object.

Select **Objects** in **Category** list and **Simple Object** in **Objects** list. Click **Next** button.

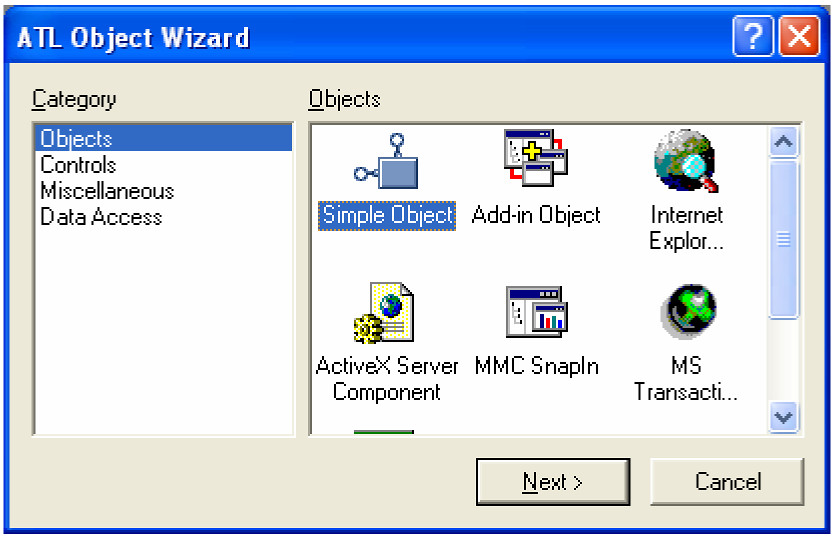
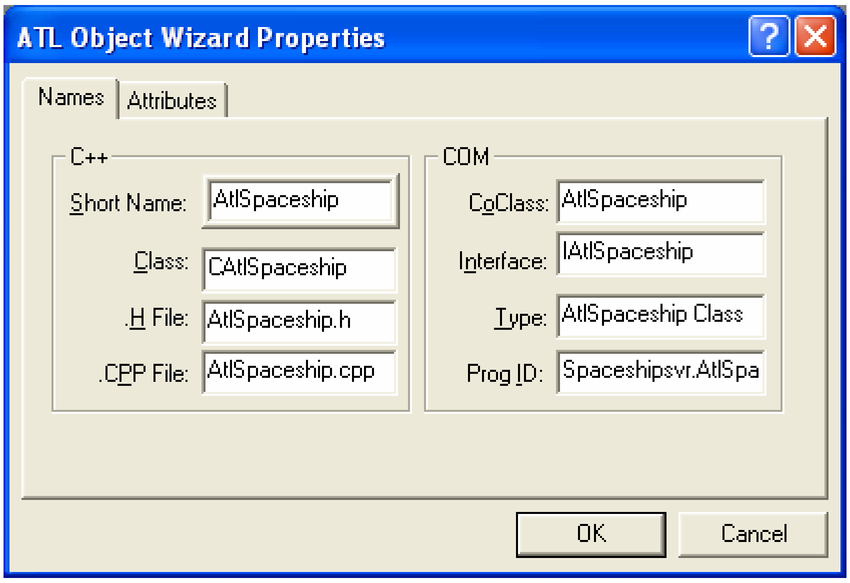


Figure 5: Selecting a **Simple Object**.

Type in **AtlSpaceship** in **Short Name** field. Click the **Attributes** tab.



C++

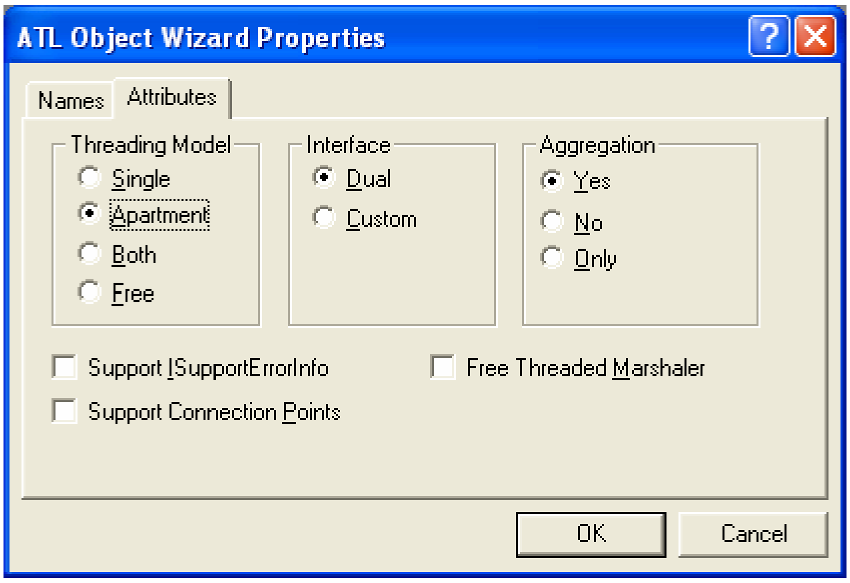
Short Name:	AtlSpaceship
Class:	CAtlSpaceship
.H File:	AtlSpaceship.h
.CPP File:	AtlSpaceship.cpp

COM

CoClass:	AtlSpaceship
Interface:	IAtlSpaceship
Type:	AtlSpaceship Class
Prog ID:	Spaceshipsvr.AtlSpa

Figure 6: Entering the ATL object information (just enter the short name field).

Leave as default for the **Attributes** options.


ATL Object Wizard Properties

Names | Attributes

Threading Model: Single, [✓ Apartment], Both, Free

Interface: [✓ Dual], Custom

Aggregation: [✓ Yes], No, Only

☐ Support ISupportErrorInfo ☐ Free Threaded Marshaler

☐ Support Connection Points

OK Cancel

Figure 7: ATL object **Attributes** options.

Using ClassView, add the following member variables to `CAtlSpaceship` class.

```
int m_nPosition;
int m_nAcceleration;
int m_nColor;
```

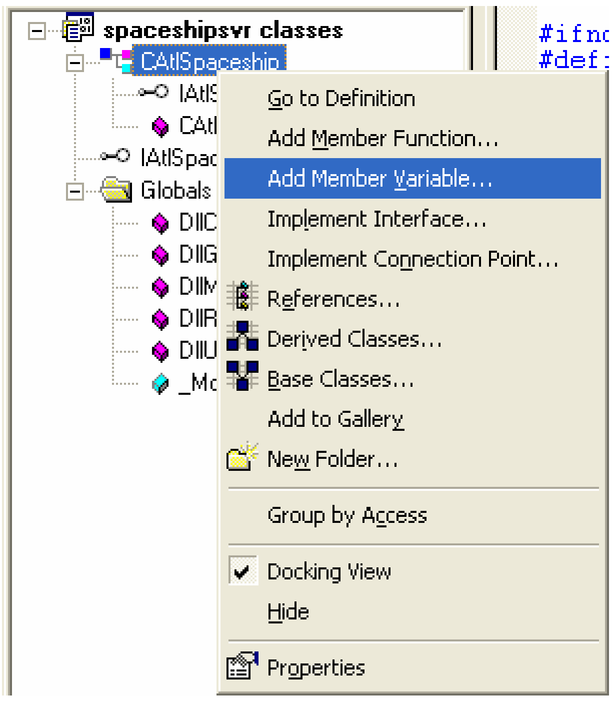


Figure 8: Adding member variable to `CAtlSpaceship` class.

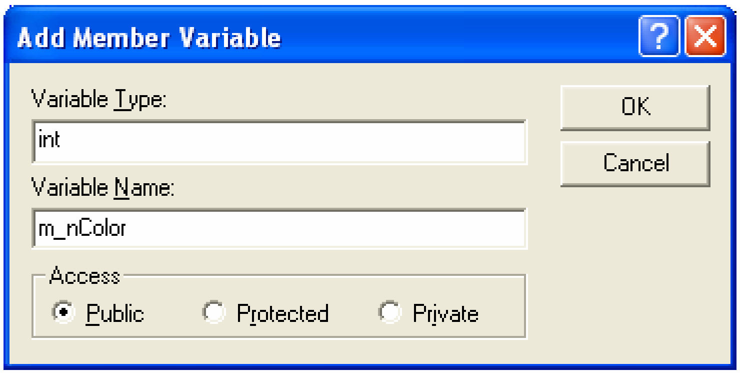


Figure 9: Adding `m_nColor` member variable.

```
// IAtlSpaceship
public:
    int m_nPosition;
    int m_nAcceleration;
    int m_nColor;

};

#endif //__ATLSPACESHIP_H_
```

Listing 1.

Initialize those variables.

```
      public:
            CAtlSpaceShip()
            {
                  m_nPosition = 0;
                  m_nAcceleration = 0;
                  m_nColor = 0;
            }
```

```
    public IDispatchImpl<IAtlSpaceship, &IID_
{
public:
    CAtlSpaceship()
    {
        m_nPosition = 0;
        m_nAcceleration = 0;
        m_nColor = 0;
    }

DECLARE_REGISTRY_RESOURCEID(IDR_ATLSPACESHIP)
```

Listing 2.

Add the following method to `IAtlSpaceship` interface.

```
[in]float fStarDate, [out, retval]BSTR* pbstrRecipient
```

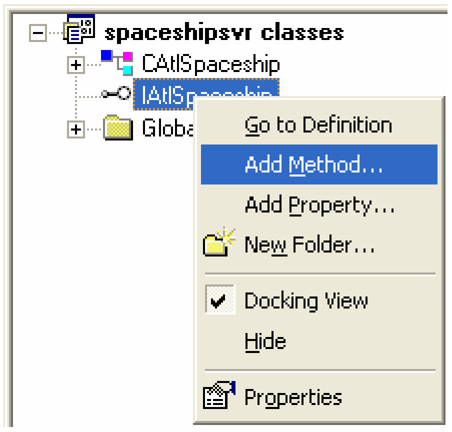
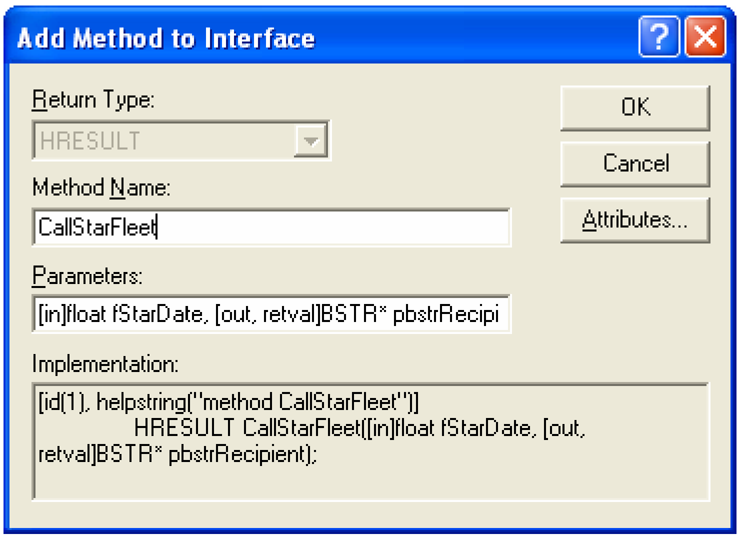


Figure 10: Adding method to `IAtlSpaceship` interface.



Return Type: HRESULT

Method Name: CallStarFleet

Parameters: [in]float fStarDate, [out, retval]BSTR* pbstrRecipi

Implementation:

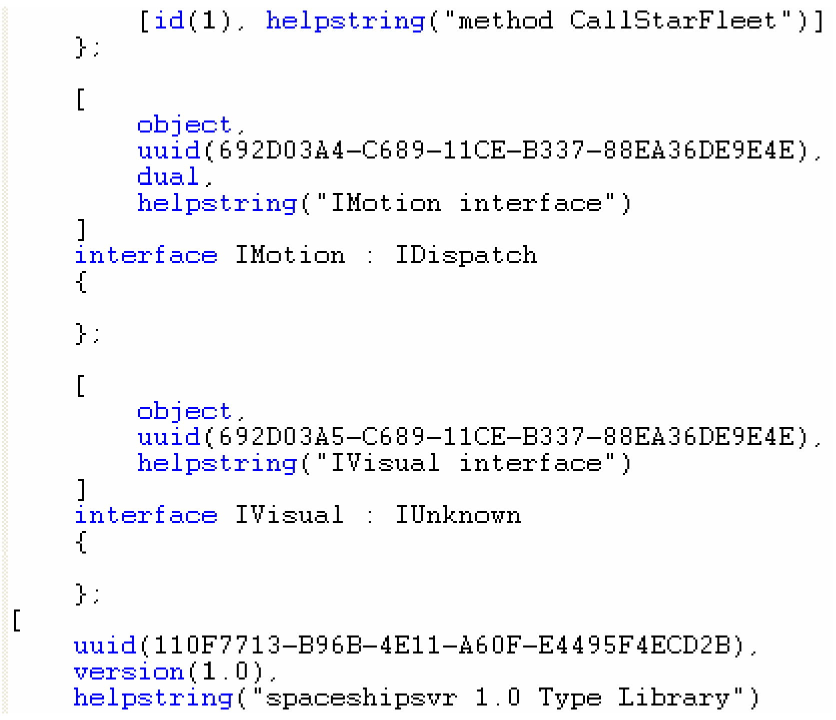
```
[id(1), helpstring("method CallStarFleet")]
        HRESULT CallStarFleet([in]float fStarDate, [out,
retval]BSTR* pbstrRecipient);
```

Figure 11: Entering method’s information.

Manually add the `IMotion` and `IVisible` interfaces to the project and to the class. Manually type the following interface definitions template in the IDL file (**spaceshipsvr.idl**) just after the:

```
interface IAtlSpaceship : IDispatch
{
        ...
        ...
};

     [
        object,
        uuid(692D03A4-C689-11CE-B337-88EA36DE9E4E),
        dual,
        helpstring("IMotion interface")
     ]
     interface IMotion : IDispatch
     {
     };

     [
        object,
        uuid(692D03A5-C689-11CE-B337-88EA36DE9E4E),
        helpstring("IVisual interface")
     ]
```

```
interface IVisual : IUnknown
{
};
```



```
        [id(1), helpstring("method CallStarFleet")]
    };

    [
        object,
        uuid(692D03A4-C689-11CE-B337-88EA36DE9E4E),
        dual,
        helpstring("IMotion interface")
    ]
    interface IMotion : IDispatch
    {

    };

    [
        object,
        uuid(692D03A5-C689-11CE-B337-88EA36DE9E4E),
        helpstring("IVisual interface")
    ]
    interface IVisual : IUnknown
    {

    };
[
    uuid(110F7713-B96B-4E11-A60F-E4495F4ECD2B),
    version(1.0),
    helpstring("spaceshipsvr 1.0 Type Library")
```

Listing 3.

Next, compile the IDL file to re-build **spaceshipsvr.h**.

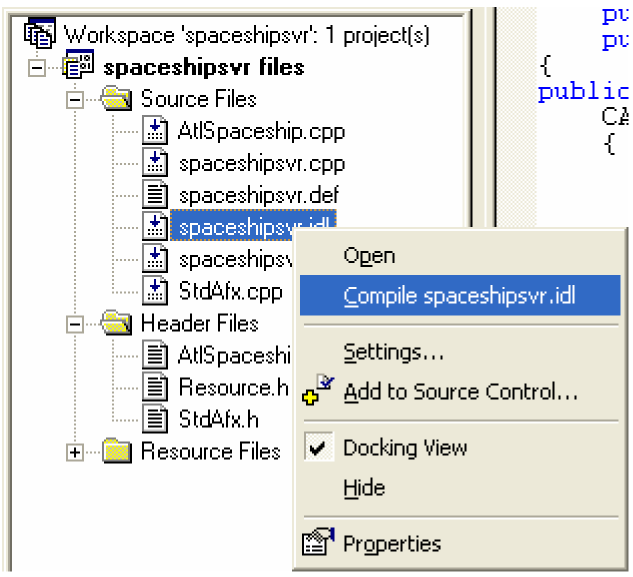
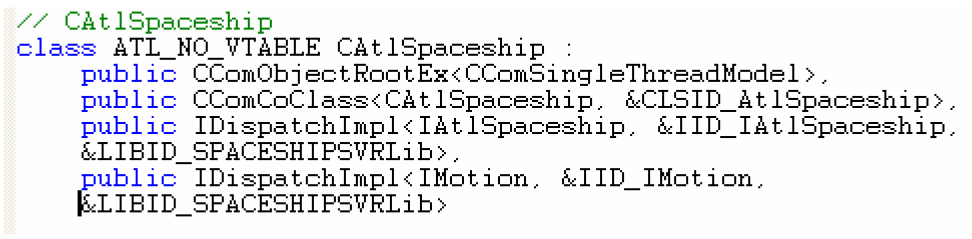


Figure 12: Compiling the IDL file.

Add `IMotion` interface to the `CSpaceship` class. Just use the `IDispatchImpl` template to provide an implementation of a dual interface. Add the `IMotion` interface as shown below.

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                              &LIBID_SPACESHIPSVRLib>,
      public IDispatchImpl<IMotion, &IID_IMotion,
                              &LIBID_SPACESHIPSVRLib>
{...};
```

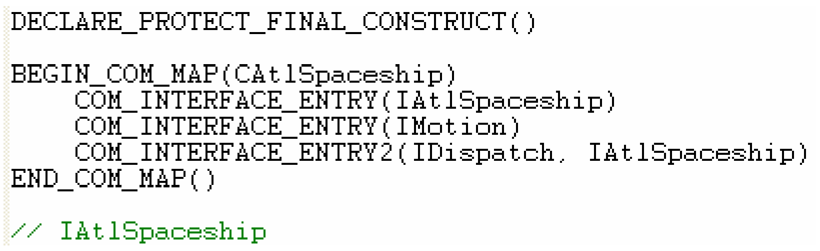


```
// CAtlSpaceship
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
    &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
    &LIBID_SPACESHIPSVRLib>
```

Listing 4.

Then, add the interface map for `IMotion`. The macro handling multiple dispatch interfaces in an ATL-based COM class is named `COM_INTERFACE_ENTRY2`. To get `QueryInterface()` working correctly, all you need to do is decide which version of `IDispatch` the client should get when asking for `IDispatch`, so add something like this for `IMotion` interface map.

```
BEGIN_COM_MAP(CAtlSpaceShip)
      COM_INTERFACE_ENTRY(IAtlSpaceShip)
      COM_INTERFACE_ENTRY(IMotion)
      COM_INTERFACE_ENTRY2(IDispatch, IAtlSpaceShip)
END_COM_MAP()
```

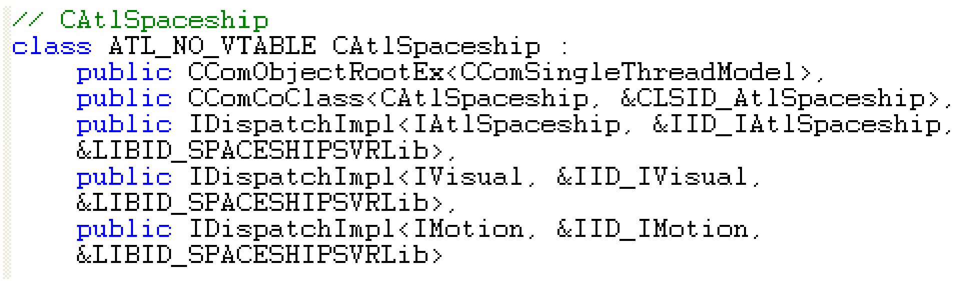


```
DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CAtlSpaceship)
    COM_INTERFACE_ENTRY(IAtlSpaceship)
    COM_INTERFACE_ENTRY(IMotion)
    COM_INTERFACE_ENTRY2(IDispatch, IAtlSpaceship)
END_COM_MAP()

// IAtlSpaceship
```

Listing 5.

Repeat the last two steps for `IVisual`. Add the interface as shown below.

```
class ATL_NO_VTABLE CAtlSpaceShip :
      public CComObjectRootEx<CComSingleThreadModel>,
      public CComCoClass<CAtlSpaceShip, &CLSID_AtlSpaceShip>,
      public IDispatchImpl<IAtlSpaceShip, &IID_IAtlSpaceShip,
                              &LIBID_SPACESHIPSVRLib>,
      public IDispatchImpl<IVisual, &IID_IVisual,
                              &LIBID_SPACESHIPSVRLib>,
      public IDispatchImpl<IMotion, &IID_IMotion,
                              &LIBID_SPACESHIPSVRLib>
```

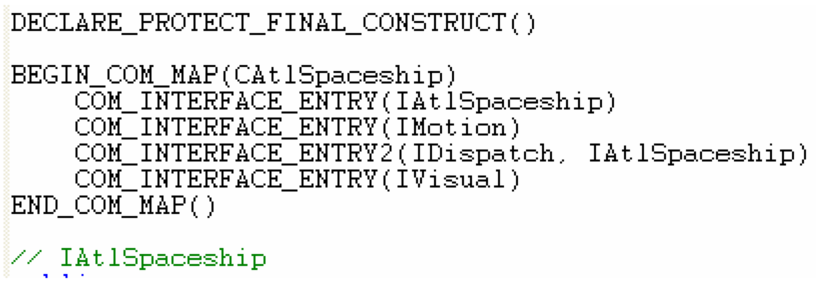


```
// CAtlSpaceship
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
    &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IVisual, &IID_IVisual,
    &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
    &LIBID_SPACESHIPSVRLib>
```

Listing 6.

Then add the interface map.

```
BEGIN_COM_MAP(CAtlSpaceShip)
      COM_INTERFACE_ENTRY(IAtlSpaceShip)
      COM_INTERFACE_ENTRY(IMotion)
      COM_INTERFACE_ENTRY2(IDispatch, IAtlSpaceship)
      COM_INTERFACE_ENTRY(IVisual)
END_COM_MAP()
```



```
DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CAtlSpaceship)
    COM_INTERFACE_ENTRY(IAtlSpaceship)
    COM_INTERFACE_ENTRY(IMotion)
    COM_INTERFACE_ENTRY2(IDispatch, IAtlSpaceship)
    COM_INTERFACE_ENTRY(IVisual)
END_COM_MAP()

// IAtlSpaceship
```

Listing 7.

Using ClassView, add the following methods to the interfaces.

Method	Interface
`Fly()`	`IMotion`
`GetPosition([out,retval]long* nPosition)`	`IMotion`
`Display()`	`IVisual`

Table 1.

Select appropriate interface and right click mouse. Select **Add Method** menu.

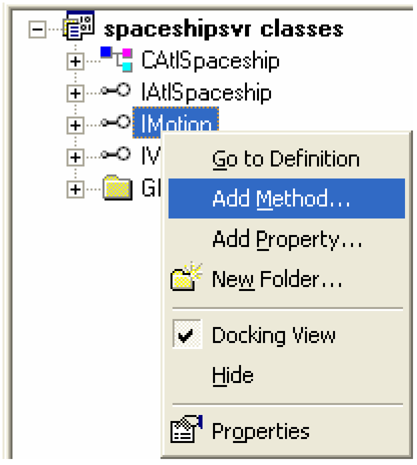
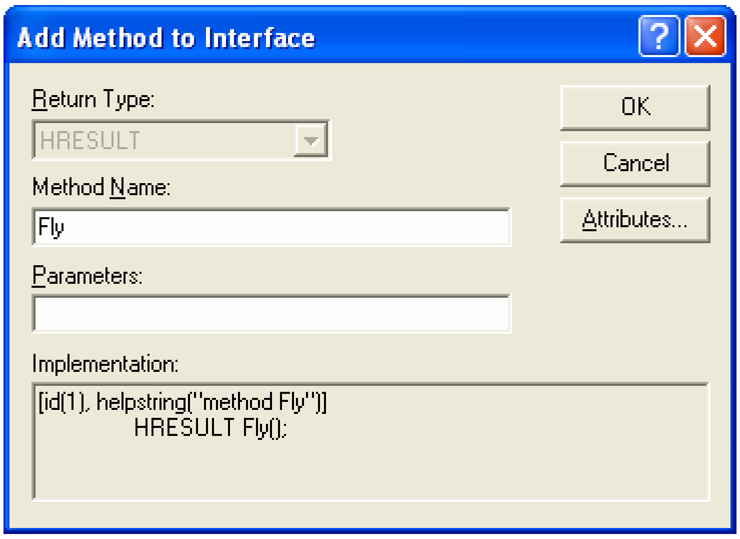


Figure 13: Adding methods to interfaces.



Return Type: HRESULT

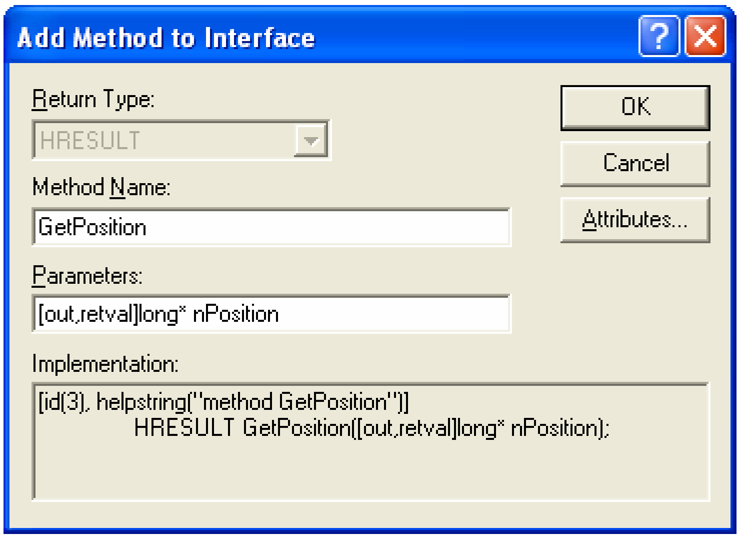
Method Name: Fly

Parameters:

Implementation:

```
[id(1), helpstring("method Fly")]
            HRESULT Fly();
```

Figure 14: Entering `Fly()` information.



Return Type: HRESULT

Method Name: GetPosition

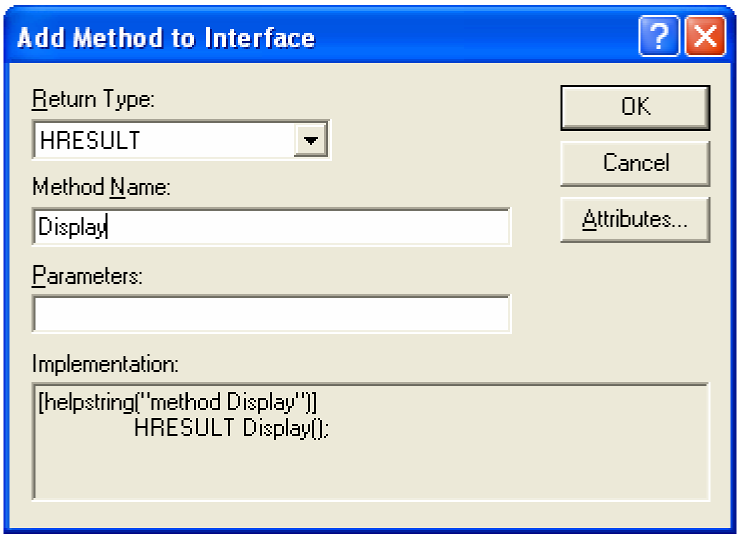
Parameters: [out,retval]long* nPosition

Implementation:

```
[id(3), helpstring("method GetPosition")]
            HRESULT GetPosition([out,retval]long* nPosition);
```

Figure 15: Entering `GetPosition()` information.

Highlight the `IVisual` interface, add the following method.



Return Type: HRESULT

Method Name: Display

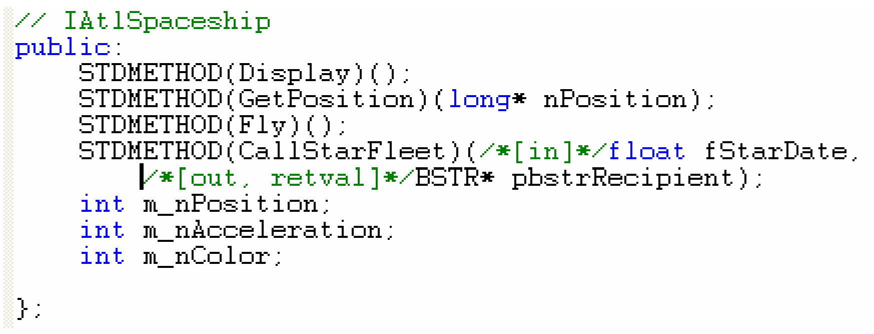
Parameters:

Implementation:

```
[helpstring("method Display")]
            HRESULT Display();
```

Figure 16: Entering `Display()` information.

You can verify the methods addition in AtlSpaceship.h file

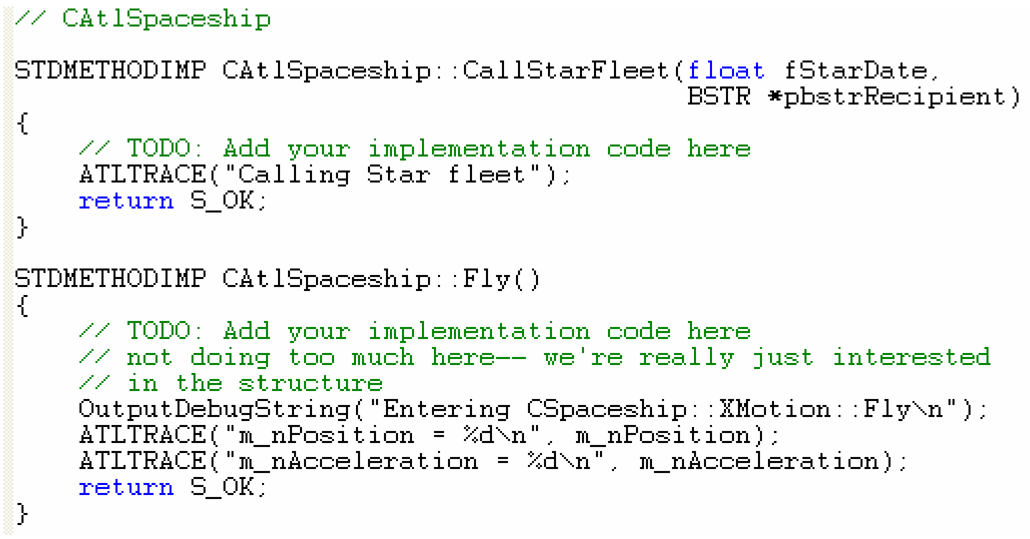


```
// IAtlSpaceship
public:
    STDMETHOD(Display)();
    STDMETHOD(GetPosition)(long* nPosition);
    STDMETHOD(Fly)();
    STDMETHOD(CallStarFleet)(/*[in]*/float fStarDate,
        /*[out, retval]*/BSTR* pbstrRecipient);
    int m_nPosition;
    int m_nAcceleration;
    int m_nColor;

};
```

Listing 8.

Finally, add the following implementation in the **AtlSpaceship.cpp** file.

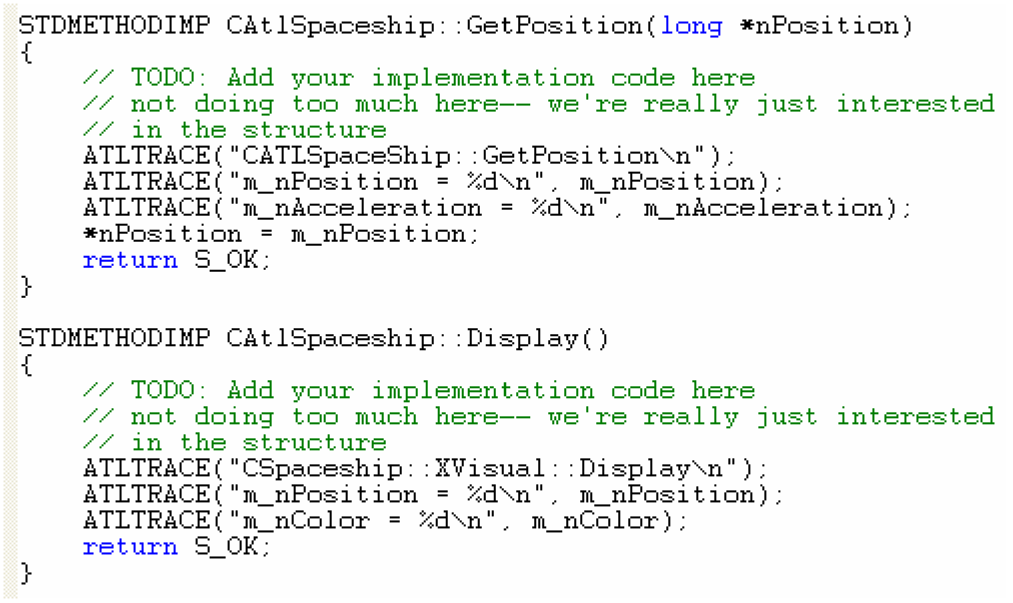
```
STDMETHODIMP CAtlSpaceship::CallStarFleet(float fStarDate, BSTR
*pbstrRecipient)
{
      // TODO: Add your implementation code here
      ATLTRACE("Calling Star fleet");
      return S_OK;
}

STDMETHODIMP CAtlSpaceship::Fly()
{
    // TODO: Add your implementation code here
    // not doing too much here-- we're really just interested in the
structure
    OutputDebugString("Entering CSpaceship::XMotion::Fly\n");
    ATLTRACE("m_nPosition = %d\n", m_nPosition);
    ATLTRACE("m_nAcceleration = %d\n", m_nAcceleration);
      return S_OK;
}
```



```
// CAtlSpaceship

STDMETHODIMP CAtlSpaceship::CallStarFleet(float fStarDate,
                                          BSTR *pbstrRecipient)
{
    // TODO: Add your implementation code here
    ATLTRACE("Calling Star fleet");
    return S_OK;
}

STDMETHODIMP CAtlSpaceship::Fly()
{
    // TODO: Add your implementation code here
    // not doing too much here-- we're really just interested
    // in the structure
    OutputDebugString("Entering CSpaceship::XMotion::Fly\n");
    ATLTRACE("m_nPosition = %d\n", m_nPosition);
    ATLTRACE("m_nAcceleration = %d\n", m_nAcceleration);
    return S_OK;
}
```

Listing 9.

```
STDMETHODIMP CAtlSpaceship::GetPosition(long *nPosition)
{
    // TODO: Add your implementation code here
    // not doing too much here-- we're really just interested in the
structure
    ATLTRACE("CATLSpaceShip::GetPosition\n");
    ATLTRACE("m_nPosition = %d\n", m_nPosition);
    ATLTRACE("m_nAcceleration = %d\n", m_nAcceleration);
      *nPosition = m_nPosition;
      return S_OK;
}

STDMETHODIMP CAtlSpaceship::Display()
{
    // TODO: Add your implementation code here
    // not doing too much here-- we're really just interested in the
structure
    ATLTRACE("CSpaceship::XVisual::Display\n");
    ATLTRACE("m_nPosition = %d\n", m_nPosition);
    ATLTRACE("m_nColor = %d\n", m_nColor);
      return S_OK;
}
```



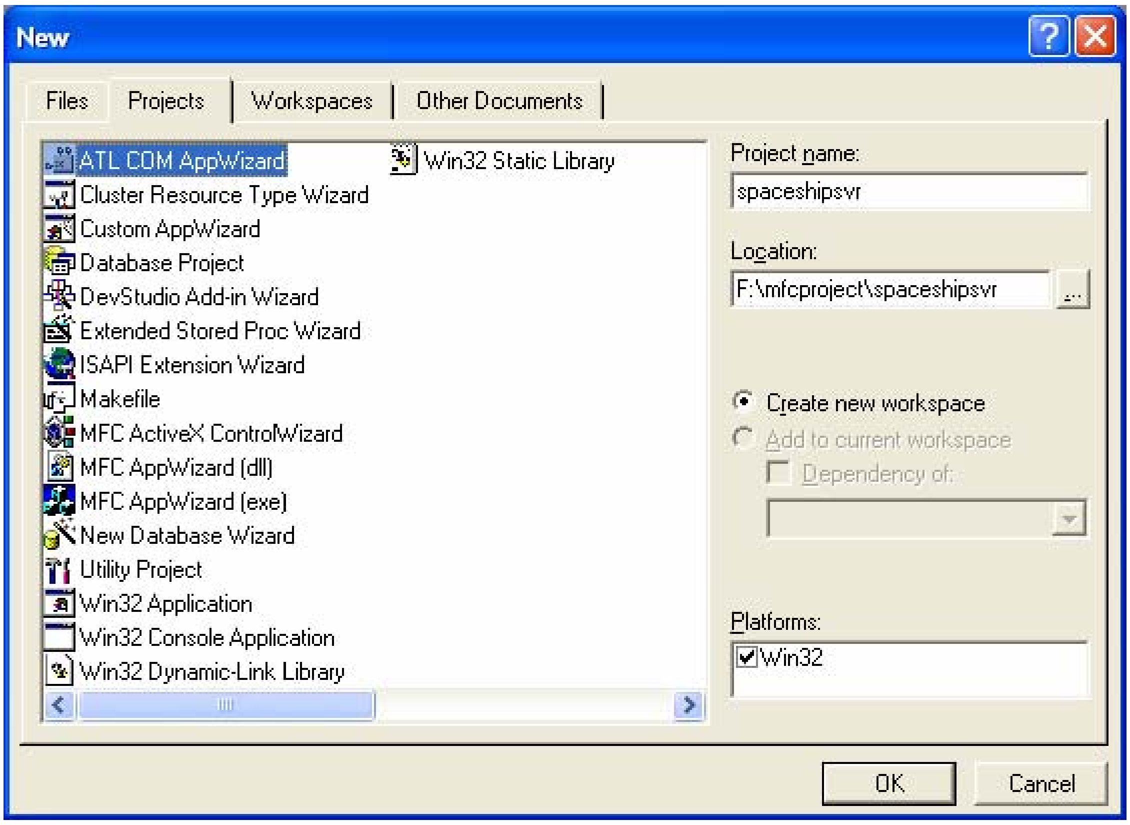
```
STDMETHODIMP CAtlSpaceship::GetPosition(long *nPosition)
{
    // TODO: Add your implementation code here
    // not doing too much here-- we're really just interested
    // in the structure
    ATLTRACE("CATLSpaceShip::GetPosition\n");
    ATLTRACE("m_nPosition = %d\n", m_nPosition);
    ATLTRACE("m_nAcceleration = %d\n", m_nAcceleration);
    *nPosition = m_nPosition;
    return S_OK;
}

STDMETHODIMP CAtlSpaceship::Display()
{
    // TODO: Add your implementation code here
    // not doing too much here-- we're really just interested
    // in the structure
    ATLTRACE("CSpaceship::XVisual::Display\n");
    ATLTRACE("m_nPosition = %d\n", m_nPosition);
    ATLTRACE("m_nColor = %d\n", m_nColor);
    return S_OK;
}
```

Listing 10.

Build **spaceshipsvr** program generating the DLL.

The Story

As always, start by selecting **New** from the **File** in Visual C++. This opens the **New** dialog with the **Projects** tab activated, as shown in Figure 17. Select **ATL COM AppWizard** from the **Projects** tab. Give your project a useful name such as **spaceshipsvr**, and click **OK**.



Project name: spaceshipsvr

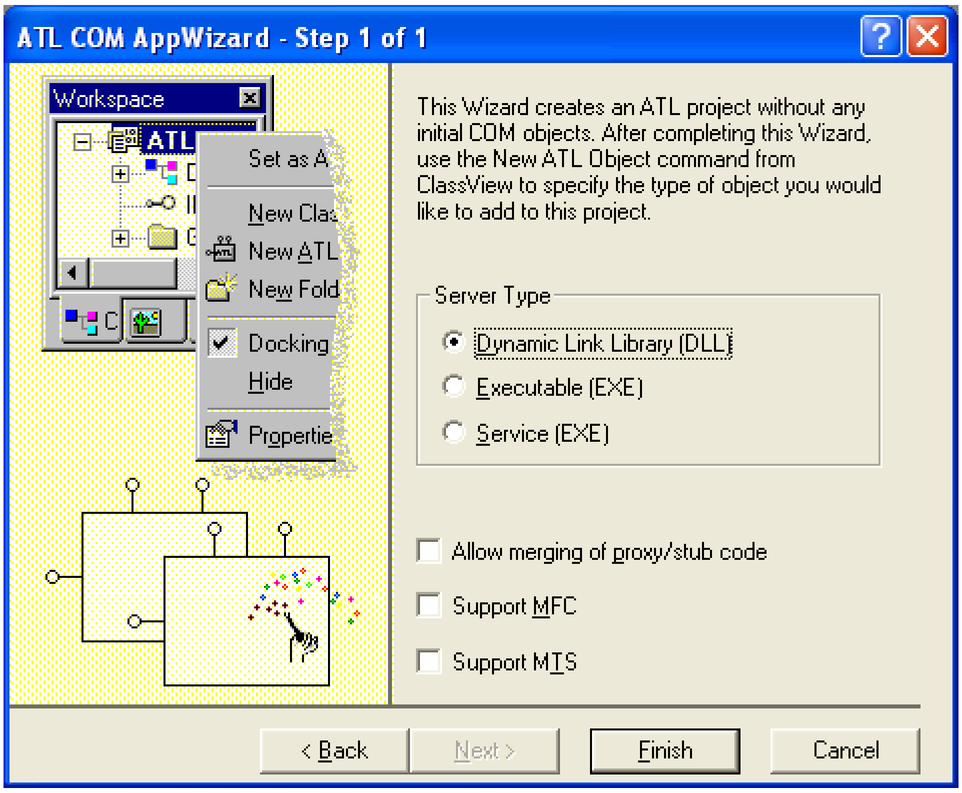
Location: F:\mfcproject\spaceshipsvr

Platforms: ✓ Win32

Figure 17: Selecting ATL COM AppWizard from the New dialog box.

ATL COM AppWizard Options

In the Step 1 dialog, shown in Figure 18, you can choose the server type for your project from a list of options. The ATL COM AppWizard gives you the choice of creating a **Dynamic Link Library (DLL)**, an **Executable (EXE)**, or a **Service (EXE)**. If you select the DLL option, the options for attaching the proxy/stub code to the DLL and for including MFC in your ATL project will be activated.


ATL COM AppWizard - Step 1 of 1

This Wizard creates an ATL project without any initial COM objects. After completing this Wizard, use the New ATL Object command from ClassView to specify the type of object you would like to add to this project.

Server Type

[✓ Dynamic Link Library (DLL)] Executable (EXE) Service (EXE)

☐ Allow merging of proxy/stub code

☐ Support MFC

☐ Support MTS

Figure 18: Step 1 of the ATL COM AppWizard.

Selecting DLL as the server type produces all the necessary pieces to make your server DLL fit into the COM environment. Among these pieces are the following well-known COM functions: `DllGetClassObject`, `DllCanUnloadNow`, `DllRegisterServer`, and `DllUnregisterServer`. Also included are the correct server lifetime mechanisms for a DLL.

If you decide you might want to run your DLL out of process as a surrogate, selecting the **Allow Merging Of Proxy/Stub Code** option permits you to package all your components into a single binary file. Proxy/stub code has traditionally shipped as a separate DLL. That way you have to distribute only a single DLL. If you decide you absolutely must include MFC in your DLL, go ahead and select the **Support MFC** check box. MFC support includes **AfxWin.h** and **AfxDisp.h** in your **StdAfx.h** file and links your project to the current version of MFC's import library. While using MFC can be very convenient and almost addictive at times, beware of dependencies you're inheriting when you include MFC. You can also select **Support MTS** to add support for Microsoft Transaction Server.

If you elect to produce an **Executable EXE** server, the ATL COM AppWizard produces code that compiles to an EXE file. The EXE will correctly register the class objects with the operating system by using `CoRegisterClassObject()` and `CoRevokeClassObject()`. The project will also insert the correct code for managing the lifetime of the executable server. Finally, if you choose the **Service EXE** option, the ATL COM AppWizard adds the necessary service-oriented code.

Using the ATL COM AppWizard to write a lightweight COM server yields several products. First, you get a project file for compiling your object. The project file ties together all the source code for the project and maintains the proper build instructions for each of the files. Second, you get some boilerplate **Interface Definition Language** (IDL) code. The IDL file is important because as the starting point for genuine COM development, it's one of the primary files you'll focus on when writing COM classes.

IDL is a purely **declarative language for describing COM interfaces**. Once a COM interface is described in an IDL file, a simple pass though the **Microsoft Interface Definition Language** (MIDL) compiler creates several more useful products. These products include:

- The pure abstract base classes needed to write COM classes.
- A type library.
- Source code for building the proxy stub DLL (necessary for standard COM remoting).

Creating a COM Class

Once you've created a COM server, you're ready to start piling COM classes into the server. Fortunately, there's an easy way to do that with the ATL Object Wizard, shown in Figure 20. Select **New ATL Object** from the **Insert** menu to start the ATL Object Wizard.
Using the ATL Object Wizard to generate a new object adds a C++ source file and a header file containing the new class definition and implementation to your project. In addition, the ATL Object Wizard adds an interface to the IDL code. Although the ATL Object Wizard takes care of pumping out a skeleton IDL file, you'll still need to understand IDL to some extent if you want to write effective COM interfaces (as you'll soon see).

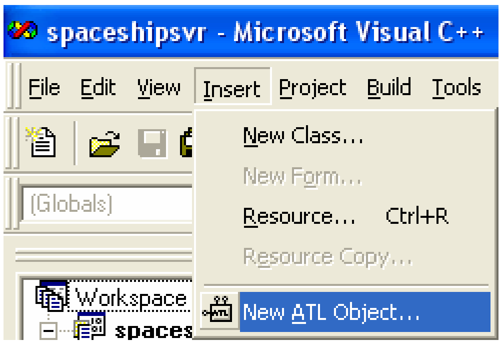


Figure 19: Adding new ATL object to project.

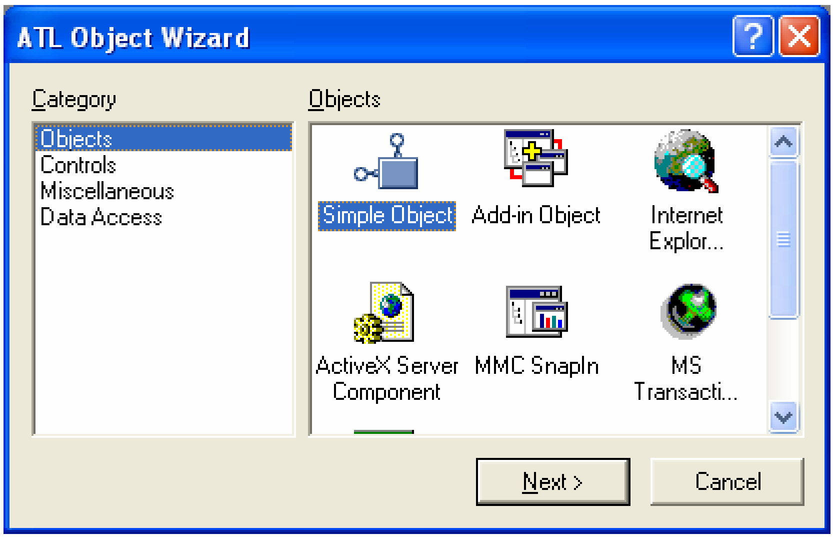
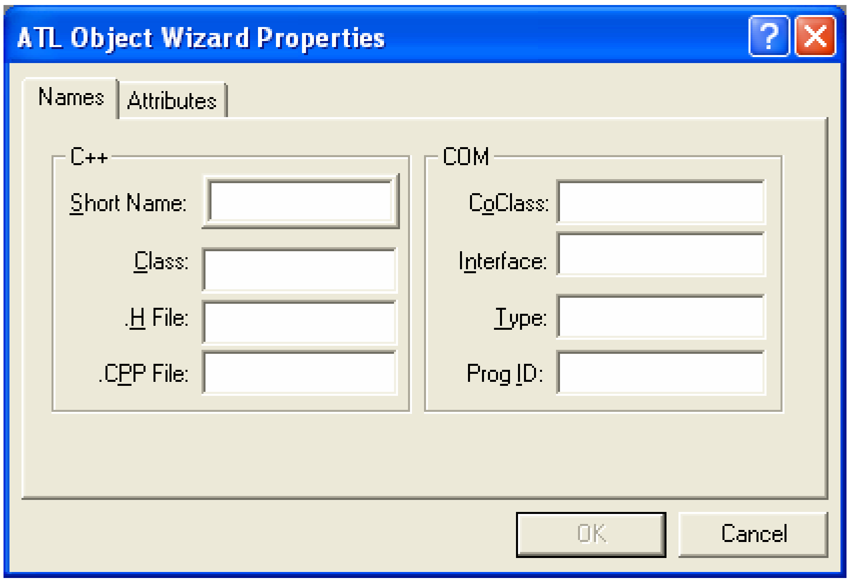


Figure 20: Using the ATL Object Wizard to insert a new ATL-based COM class into the project.

After you choose the type of ATL object, click **Next** to display the **ATL Object Wizard Properties** dialog.


ATL Object Wizard Properties

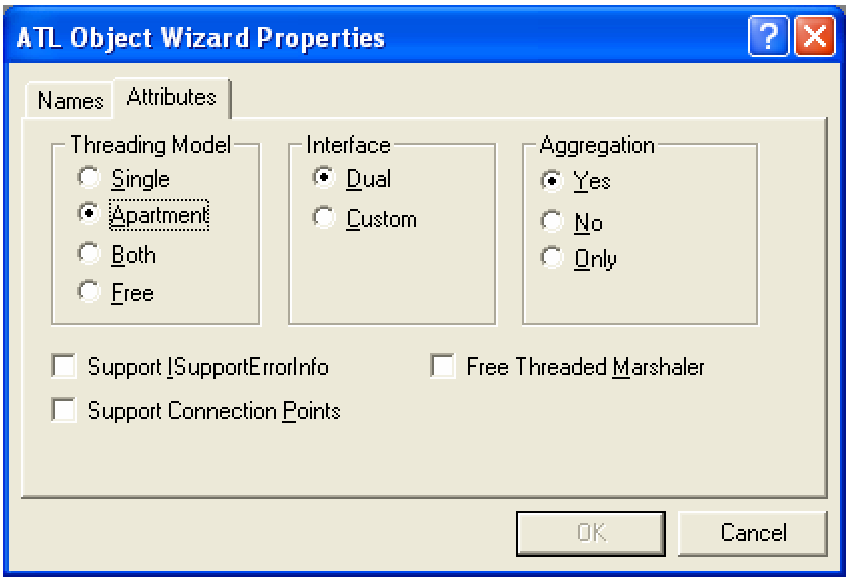
Names | Attributes

C++
Short Name:
Class:
.H File:
.CPP File:

COM
CoClass:
Interface:
Type:
Prog ID:

OK | Cancel

Figure 21: ATL object's Name page.


ATL Object Wizard Properties

Names | Attributes

Threading Model: Single [✓ Apartment] Both Free

Interface: [✓ Dual] Custom

Aggregation: [✓ Yes] No Only

Support ISupportErrorInfo

Support Connection Points

Free Threaded Marshaler

OK | Cancel

Figure 22: ATL object attribute page.

Depending on which object you choose, the **Attributes** tab of the ATL Object Wizard Properties dialog allows you to select the threading model for your COM class, and whether you want a dual (`IDispatch`-based) or a custom interface. The dialog also allows you to choose how your class will support aggregation. In addition, the Object Wizard lets you easily include the `ISupportErrorInfo` interface and connection points in your class. Finally, you can aggregate to the **Free-Threaded Marshaler** if you so choose.

Apartments and Threading

To figure out COM, you have to understand that COM is centered on the notion of abstraction, hiding as much information as possible from the client. One piece of information that COM hides from the client is whether COM class is thread-safe. The client should be able to use an object as it sees fit without having to worry about whether an object properly serializes access to itself, that is, properly protects access to its internal data. COM defines the notion of an apartment to provide this abstraction.
An apartment defines an execution context, or thread, that houses interface pointers. A thread enters an apartment by calling a function from the `CoInitialize()` family: `CoInitialize()`, `CoInitializeEx()`, or `OleInitialize()`. Then COM requires that all method calls to an interface pointer be executed within the apartment that initialized the pointer (in other words, from the same thread that called `CoCreateInstance()`).

COM defines two kinds of apartments, **single-threaded apartments** and **multithreaded apartments**. Single-threaded apartments can house only one thread while multithreaded apartments can house several threads. While a process can have only one multithreaded apartment, it can have many single-threaded apartments. An apartment can house any number of COM objects.
A single-threaded apartment guarantees that COM objects created within it will have method calls serialized through the **remoting layer**, while a COM object created within a multithreaded apartment will not. A helpful way to remember the difference between apartments is to think of it this way: instantiating a COM object within the multithreaded apartment is like putting a piece of data into the global scope where multiple threads can get to it. Instantiating a COM object within a single-threaded apartment is like putting data within the scope of only one thread. The bottom line is that COM classes that want to live in the multithreaded apartment had better be thread-safe, while COM classes that are satisfied living in their own apartments need not worry about concurrent access to their data.
A COM object living within a different process space from its client has its method calls serialized automatically via the remoting layer. However, a COM object living in a DLL might want to provide its own internal protection (using critical sections, for example) rather than having the remoting layer protect it. A COM class advertises its thread safety to the world via a Registry setting. This named value lives in the Registry under the `CLSID` under `HKEY_CLASSES_ROOT` like this:

```
[HKCR\CLSID\{some GUID …}\InprocServer32]
@="C:\SomeServer.DLL"
ThreadingModel=<thread model>
```

The `ThreadingModel` can be one of four values: **Single**, **Both**, **Free**, or **Apartment**, or it can be blank. ATL provides support for all current threading models. Here's a rundown of what each value indicates:

- Single or blank indicates that the class executes in the main thread only (the first single thread created by the client).
- Both indicates that the class is thread-safe and can execute in both the single-threaded and multithreaded apartments. This value tells COM to use the same kind of apartment as the client.
- Free indicates that the class is thread-safe. This value tells COM to force the object inside the multithreaded apartment.
- Apartment indicates that the class isn't thread-safe and must live in its own single-threaded apartment.

When you choose a threading model in the ATL Object Wizard, the wizard inserts different code into your class depending upon your selection. For example, if you select the **apartment** model, the Object Wizard derives your class from `CComObjectRootEx` and includes `CComSingleThreadModel` as the template parameter like this:

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                         &LIBID_SPACESHIPSVRLib>
{...};
```

The `CComSingleThreadModel` template parameter mixes in the more efficient standard increment and decrement operations for `IUnknown` (because access to the class is automatically serialized). In addition, the ATL Object Wizard causes the class to insert the correct threading model value in the Registry. If you choose the **Single** option in the ATL Object Wizard, the class uses the `CComSingleThreadModel` but leaves the `ThreadingModel` value blank in the Registry.
Choosing **Both** or **Free** causes the class to use the `CComMultiThreadModel` template parameter, which employs the thread-safe Win32 increment and decrement operations `InterlockedIncrement` and `InterlockedDecrement`. For example, a free-threaded class definition looks like this:

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx< CComMultiThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                         &LIBID_SPACESHIPSVRLib>
{...};
```

Choosing Both for your threading model inserts Both as the data for the `ThreadingModel` value, while choosing Free uses the data value Free for the `ThreadingModel` value.

Connection Points and ISupportErrorInfo

Adding connection to your COM class is easy. Selecting the **Support Connection Points** check box causes the class to derive from `IConnectionPointImpl`. This option also adds a blank connection map to your class. Adding connection points (for example, an event set) to your class is simply a matter of performing the following four steps:

1. Define the callback interface in the IDL file.
2. Use the ATL proxy generator to create a proxy.
3. Add the proxy class to the COM class.
4. Add the connection points to the connection point map.

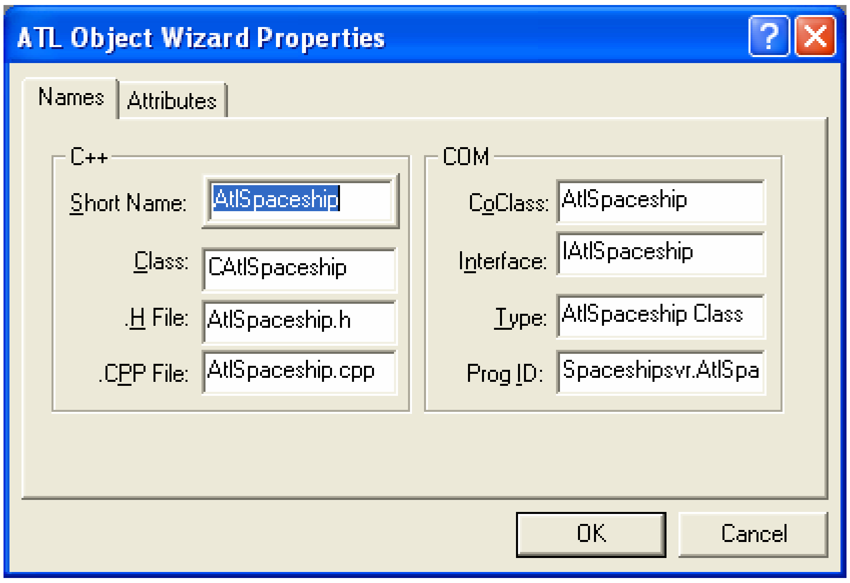
ATL also includes support for `ISupportErrorInfo`. The `ISupportErrorInfo` interface ensures that error information is propagated up the call chain correctly. OLE Automation objects that use the error-handling interfaces must implement `ISupportErrorInfo`. Selecting `Support ISupportErrorInfo` in the ATL Object Wizard dialog causes the ATL-based class to derive from `ISupportErrorInfoImpl`.

The Free-Threaded Marshaler

Selecting the **Free Threaded Marshaler** option aggregates the COM free-threaded marshaler to your class. The generated class does this by calling `CoCreateFreeThreadedMarshaler()` in its `FinalConstruct()` function. The free-threaded marshaler allows thread-safe objects to bypass the standard marshaling that occurs whenever cross-apartment interface methods are invoked, allowing threads living in one apartment to access interface methods in another apartment as though they were in the same apartment. This process speeds up cross-apartment calls tremendously. The free-threaded marshaler does this by implementing the `IMarshal` interface. When the client asks the object for an interface, the remoting layer calls `QueryInterface()`, asking for `IMarshal`. If the object implements `IMarshal` (in this case, the object implements `IMarshal` because the ATL Object Wizard also adds an entry into the class's interface to handle `QueryInterface()` requests for `IMarshal`) and the marshaling request is in process, the free-threaded marshaler actually copies the pointer into the marshaling packet. That way, the client receives an actual pointer to the object. The client talks to the object directly without having to go through proxies and stubs. Of course, if you choose the **Free Threaded Marshaler** option, all data in your object had better be thread-safe. Just be very cautious if you check this box.

Implementing the Spaceship Class Using ATL

We'll create the spaceship class using the defaults provided by the **ATL Object Wizard** in the **ATL Object Wizard Properties** dialog. For example, the spaceship class will have a dual interface, so it will be accessible from environments such as VBScript on a Web page. In addition, the spaceship class will be an apartment model object, meaning COM will manage most of the concurrency issues. The only information you need to supply to the ATL Object Wizard is a **clever name**. Enter a value such as **AtlSpaceship** in the **Short Name** text box on the **Names** tab.


ATL Object Wizard Properties

Names | Attributes

C++
Short Name: AtlSpaceship
Class: CAtlSpaceship
.H File: AtlSpaceship.h
.CPP File: AtlSpaceship.cpp

COM
CoClass: AtlSpaceship
Interface: IAtlSpaceship
Type: AtlSpaceship Class
Prog ID: Spaceshipsvr.AtlSpa

Figure 23: Entering object's name information.

You don't need to set any of the other options right now. For instance, you don't need to set the **Support Connection Points** option because we'll cover connections in the next module. You can always add connection points later by typing them in by hand.
If you tell the ATL Object Wizard to create a **Simple Object COM** class named **ATLSpaceship**, here's the class definition it generates:

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                         &LIBID_SPACESHIPSVRLib>
{...};
```

While ATL includes quite a few COM-oriented C++ classes, those listed in the spaceship class's inheritance list above are enough to get a flavor of how ATL works.
The most generic ATL-based COM objects derive from three base classes: `CComObjectRoot`, `CComCoClass`, and `IDispatch`. `CComObjectRoot` implements `IUnknown` and manages the identity of the class. This means `CComObjectRoot` implements `AddRef()` and `Release()` and hooks into ATL's `QueryInterface()` mechanism. `CComCoClass` manages the COM class's class object and some general error reporting. In the class definition above, CComCoClass adds the class object that knows how to create `CAtlSpaceship` objects. Finally, the code produced by the ATL Object Wizard includes an implementation of IDispatch based on the type library produced by compiling the IDL. The default `IDispatch` is based on a dual interface (which is an `IDispatch` interface followed by the functions defined in the IDL).
As you can see, using ATL to implement COM classes is different from using pure C++. The Tao of ATL differs from what you might be used to when developing normal C++ classes. With ATL, the most important part of the project is the interfaces, which are described in IDL. By adding functions to the interfaces in the IDL code, you automatically add functions to the concrete classes implementing the interfaces. The functions are added automatically because the projects are set up such that compiling the IDL file yields a C++ header file with those functions. All that's left for you to do after adding the functions in the interface is to implement those functions in the C++ class. The IDL file also provides a type library so the COM class can implement `IDispatch`. However, while ATL is useful for implementing lightweight COM services and objects, ATL is also a new means by which you can create ActiveX controls, as you'll see in the next module.

Basic ATL Architecture

If you've experimented at all with ATL, you've seen how it simplifies the process of implementing COM classes. The tool support is quite good, it's almost as easy to develop COM classes using Visual C++ 6.0 as it is to create MFC-based programs. Just use AppWizard to create a new ATL-based class. However, instead of using

ClassWizard (as you would to handle messages and to add dialog box member variables), use ClassView to add new function definitions to an interface. Then simply fill in the functions within the C++ code generated by ClassView. The code generated by AppWizard includes all the necessary code for implementing your class, including an implementation of `IUnknown`, a server module to house your COM class, and a class object that implements `IClassFactory`.
Writing COM objects as we've just described is certainly more convenient than most other methods. But exactly what happens when you use the AppWizard to generate the code for you? Understanding how ATL works is important if you want to extend your ATL-based COM classes and servers much beyond what AppWizard and ClassView provide. For example, ATL provides support for advanced interface techniques such as tear-off interfaces. Unfortunately, there's no Wizard option for implementing a tear-off interface. Even though ATL supports it, you've got to do a little work by hand to accomplish the tear-off interface. Understanding how ATL implements `IUnknown` is helpful in this situation. Let's examine the `CAtlSpaceship` class in a bit more detail. Here's the entire definition:

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                         &LIBID_SPACESHIPSVRLib>
{
public:
    CAtlSpaceship()
    {   }

DECLARE_REGISTRY_RESOURCEID(IDR_ATLSPACESHIP)

BEGIN_COM_MAP(CAtlSpaceship)
    COM_INTERFACE_ENTRY(IAtlSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

// IAtlSpaceship
public:
};
```

While this is ordinary vanilla C++ source code, it differs from normal everyday C++ source code for implementing a COM object in several ways. For example, while many COM class implementations derive strictly from COM interfaces, this COM class derives from several templates. In addition, this C++ class uses several odd-looking macros. As you examine the code, you'll see ATL's implementation of `IUnknown`, as well as a few other interesting topics, such as a technique for managing vtable bloat and an uncommon use for templates. Let's start by taking a look at the first symbol in the wizard-generated macro code: `ATL_NO_VTABLE`.

Managing VTBL Bloat

COM interfaces are easily expressed in C++ as pure abstract base classes. Writing COM classes using multiple inheritance (there are other ways to write COM classes) is merely a matter of adding the COM interface base classes to your inheritance list and implementing the union of all the functions. Of course, this means that the memory footprint of your COM server will include a significant amount of vtable overhead for each interface implemented by your class. That's not a big deal if you have only a few interfaces and your C++ class hierarchy isn't very deep. However, implementing interfaces this way does add overhead that tends to accumulate as interfaces are added and hierarchies deepen. ATL provides a way to cut down on some of the overhead introduced by a lot of virtual functions. ATL defines the following symbol:

```
#define ATL_NO_VTABLE  __declspec(novtable)
```

Using `ATL_NO_VTABLE` prevents an object's vtable (vtable) from being initialized in the constructor, thereby eliminating from the linker the vtable and all the functions pointed to by the vtable for that class. This elimination can lower the size of your COM server somewhat, provided the most-derived class does not use the `novtable declspec` shown above. You'll notice the size difference in classes with deep derivation lists. One caveat, however: calling virtual functions from the constructor of any object that uses this `declspec` is unsafe because vptr is uninitialized. The second line in the class declaration previously shown demonstrates that CAtlSpaceship derives from CComObjectRootEx. This is where you get to ATL's version of IUnknown.

ATL's IUnknown: CComObjectRootEx

While CComObjectRootEx isn't quite at the top of the ATL hierarchy, it's pretty close. The actual base class for a COM object in ATL is a class named CComObjectRootBase. (Both class definitions are located in ATLCOM.H.) Looking at CComObjectRootBase reveals the code you might expect for a C++ based COM class. ComObjectRootBase includes a DWORD member named m_dwRef for reference counting. You'll also see OuterAddRef, OuterRelease, and OuterQueryInterface to support COM aggregation and tear-off interfaces. Looking at CComObjectRootEx reveals InternalAddRef(), InternalRelease(), and InternalQueryInterface() for performing the regular native reference counting, and QueryInterface() mechanisms for class instances with object identity.
Notice that CAtlSpaceship's definition shows that the class is derived from CComObjectRootEx and that CComObjectRootEx is a parameterized template class. The listing below shows the definition of CComObjectRootEx.

```
template <class ThreadModel>
class CComObjectRootEx : public CComObjectRootBase
{
public:
    typedef ThreadModel _ThreadModel;
    typedef _ThreadModel::AutoCriticalSection _CritSec;
    typedef CComObjectLockT<_ThreadModel> ObjectLock;

    ULONG InternalAddRef()
    {
        ATLASSERT(m_dwRef != -1L);
        return _ThreadModel::Increment(&m_dwRef);
    }
    ULONG InternalRelease()
    {
        ATLASSERT(m_dwRef > 0);
        return _ThreadModel::Decrement(&m_dwRef);
    }

    void Lock() {m_critsec.Lock();}
    void Unlock() {m_critsec.Unlock();}
private:
    _CritSec m_critsec;
};
```

CComObjectRootEx is a template class that varies in type based on the kind of threading model class passed in as the template parameter. In fact, ATL supports several threading models: **Single-Threaded Apartments** (STAs), **Multi-Threaded Apartments** (MTAs), and **Free Threading**. ATL includes three preprocessor symbols for selecting the various default threading models for your project: _ATL_SINGLE_THREADED, _ATL_APARTMENT_THREADED, and _ATL_FREE_THREADED.
Defining the preprocessor symbol _ATL_SINGLE_THREADED in **stdafx.h** changes the default threading model to support only one STA-based thread. This option is useful for out-of-process servers that don't create any extra threads. Because the server supports only one thread, ATL's global state can remain unprotected by critical sections and the server is therefore more efficient. The downside is that your server can support only one thread. Defining _ATL_APARTMENT_THREADED for the preprocessor causes the default threading model to support multiple STA-based threads. This is useful for apartment model in-process servers (servers supporting the ThreadingModel=Apartment Registry value). Because a server employing this threading model can support multiple threads, ATL protects its global state using **critical sections**. Finally, defining the _ATL_FREE_THREADED preprocessor symbol creates servers compatible with any threading environment. That is, ATL protects its global state using critical sections, and each object in the server will have its own critical sections to maintain data safety.

These preprocessor symbols merely determine which threading class to plug into `CComObjectRootEx` as a template parameter. ATL provides three threading model classes. The classes provide support for the most efficient yet thread-safe behavior for COM classes within each of the three contexts listed above. The three classes are `CComMultiThreadModelNoCS`, `CComMultiThreadModel`, and `CComSingleThreadModel`. The following listing shows the three threading model classes within ATL:

```
class CComMultiThreadModelNoCS
{
public:
    static ULONG WINAPI Increment(LPLONG p)
                        {return InterlockedIncrement(p);}
    static ULONG WINAPI Decrement(LPLONG p)
                        {return InterlockedDecrement(p);}
    typedef CComFakeCriticalSection AutoCriticalSection;
    typedef CComFakeCriticalSection CriticalSection;
    typedef CComMultiThreadModelNoCS ThreadModelNoCS;
};

class CComMultiThreadModel
{
public:
    static ULONG WINAPI Increment(LPLONG p)
                        {return InterlockedIncrement(p);}
    static ULONG WINAPI Decrement(LPLONG p)
                        {return InterlockedDecrement(p);}
    typedef CComAutoCriticalSection AutoCriticalSection;
    typedef CComCriticalSection CriticalSection;
    typedef CComMultiThreadModelNoCS ThreadModelNoCS;
};

class CComSingleThreadModel
{
public:
    static ULONG WINAPI Increment(LPLONG p) {return ++(*p);}
    static ULONG WINAPI Decrement(LPLONG p) {return --(*p);}
    typedef CComFakeCriticalSection AutoCriticalSection;
    typedef CComFakeCriticalSection CriticalSection;
    typedef CComSingleThreadModel ThreadModelNoCS;
};
```

Notice that each of these classes exports two static functions, `Increment()` and `Decrement()` and various aliases for critical sections.

`CComMultiThreadModel` and `CComMultiThreadModelNoCS` both implement `Increment()` and `Decrement()` using the thread-safe Win32 `InterlockedIncrement()` and `InterlockedDecrement()` functions. `CComSingleThreadModel` implements `Increment()` and `Decrement()` using the more conventional `++` and `--` operators.

In addition to implementing incrementing and decrementing differently, the three threading models also manage critical sections differently. ATL provides wrappers for two critical sections - a `CComCriticalSection` (which is a plain wrapper around the Win32 critical section API) and `CComAutoCriticalSection` (which is the same as `CComCriticalSection` with the addition of automatic initialization and cleanup of critical sections). ATL also defines a "fake" critical section class that has the same binary signature as the other critical section classes but doesn't do anything. As you can see from the class definitions, `CComMultiThreadModel` uses real critical sections while `CComMultiThreadModelNoCS` and `CComSingleThreadModel` use the fake no-op critical sections.

So now the minimal ATL class definition makes a bit more sense. `CComObjectRootEx` takes a thread model class whenever you define it. `CAtlSpaceship` is defined using the `CComSingleThreadModel` class, so it uses the `CComSingleThreadModel` methods for incrementing and decrementing as well as the fake no-op critical sections. Thus `CAtlSpaceship` uses the most efficient behavior since it doesn't need to worry about protecting data. However, you're not stuck with that model. If you want to make `CAtlSpaceship` safe for any threading environment, for example, simply redefine `CAtlSpaceship` to derive from `CComObjectRootEx` using CComMultiThreadModel as the template parameter. AddRef() and Release() calls are automatically mapped to the correct Increment() and Decrement() functions.

ATL and QueryInterface()

It looks as though ATL took a cue from MFC for implementing QueryInterface(), ATL uses a lookup table just like MFC's version. Take a look at the middle of CAtlSpaceship's definition, you'll see a construct based on macros called the interface map. ATL's interface maps constitute its QueryInterface() mechanism.
Clients use QueryInterface() to arbitrarily widen the connection to an object. That is, when a client needs a new interface, it calls QueryInterface() through an existing interface. The object then looks at the name of the requested interface and compares that name to all the interfaces implemented by the object. If the object implements the interface, the object hands the interface back to the client. Otherwise, QueryInterface() returns an error indicating that no interface was found.
Traditional QueryInterface() implementations usually consist of long if-then statements. For example, a standard implementation of QueryInterface() for a multiple-inheritance COM class might look like this:

```
class CAtlSpaceship: public IDispatch, IAtlSpaceship
{
    HRESULT QueryInterface(RIID riid, void** ppv)
    {
        if(riid == IID_IDispatch)
            *ppv = (IDispatch*) this;
        else if(riid == IID_IAtlSpaceship || riid == IID_IUnknown)
            *ppv = (IAtlSpaceship *) this;
        else
        {
            *ppv = 0;
            return E_NOINTERFACE;
        }

        ((IUnknown*)(*ppv))->AddRef();
        return NOERROR;
    }
    // AddRef, Release, and other functions
};
```

As you'll see in a moment, ATL uses a lookup table instead of this conventional if-then statement. ATL's lookup table begins with a macro named BEGIN_COM_MAP. The listing below shows the full definition of BEGIN_COM_MAP.

```
#define BEGIN_COM_MAP(x) public:
    typedef x _ComMapClass;
    static HRESULT WINAPI _Cache(void* pv,
                                 REFIID iid,
                                 void** ppvObject,
                                 DWORD dw)
   {
        _ComMapClass* p = (_ComMapClass*)pv;
        p->Lock();
        HRESULT hRes = CComObjectRootBase::_Cache(pv,
                                     iid,
                                     ppvObject,
                                     dw);
        p->Unlock();
        return hRes;
    }

    IUnknown* GetRawUnknown()
    {
        ATLASSERT(_GetEntries()[0].pFunc == _ATL_SIMPLEMAPENTRY);
```

```
            return (IUnknown*)((int)this+_GetEntries()->dw);
        }

        _ATL_DECLARE_GET_UNKNOWN(x)
        HRESULT _InternalQueryInterface(REFIID iid, void** ppvObject)
        {
            return InternalQueryInterface(this,
                                          _GetEntries(),
                                          iid,
                                          ppvObject);
        }

        const static _ATL_INTMAP_ENTRY* WINAPI _GetEntries()
        {
        static const _ATL_INTMAP_ENTRY _entries[] = {
            DEBUG_QI_ENTRY(x)
            ...
            ...
            ...
        #define END_COM_MAP()   {NULL, 0, 0}};\
        return _entries;}
```

Each class that uses ATL for implementing IUnknown specifies an interface map to provide to `InternalQueryInterface`. ATL's interface maps consist of structures containing interface ID (`GUID`)/`DWORD`/function pointer tuples. The following listing shows the type named `_ATL_INTMAP_ENTRY` that contains these tuples.

```
    struct _ATL_INTMAP_ENTRY
    {
        const IID* piid;
        DWORD dw;
        _ATL_CREATORARGFUNC* pFunc;
    };
```

The first member is the interface ID (a `GUID`), and the second member indicates what action to take when the interface is queried. There are three ways to interpret the third member. If pFunc is equal to the constant `_ATL_SIMPLEMAPENTRY` (the value 1), `dw` is an offset into the object. If `pFunc` is non-null but not equal to 1, `pFunc` indicates a function to be called when the interface is queried. If `pFunc` is NULL, `dw` indicates the end of the `QueryInterface()` lookup table.
Notice that `CAtlSpaceship` uses `COM_INTERFACE_ENTRY`. This is the interface map entry for regular interfaces. Here's the raw macro:

```
    #define offsetofclass(base, derived)
    ((DWORD)(static_cast<base*>((derived*)8))-8)

    #define COM_INTERFACE_ENTRY(x)\
        {&_ATL_IIDOF(x), \
        offsetofclass(x, _ComMapClass), \
        _ATL_SIMPLEMAPENTRY}
```

`COM_INTERFACE_ENTRY` fills the `_ATL_INTMAP_ENTRY` structure with the interface's `GUID`. In addition, notice how offsetofclass casts the `this` pointer to the right kind of interface and fills the `dw` member with that value. Finally, `COM_INTERFACE_ENTRY` fills the last field with `_ATL_SIMPLEMAPENTRY` to indicate that `dw` points to an offset into the class. For example, the interface map for `CAtlSpaceship` looks like this after the preprocessor is done with it:

```
    const static _ATL_INTMAP_ENTRY* __stdcall _GetEntries()
    {
        static const _ATL_INTMAP_ENTRY _entries[] = {
            {&IID_IAtlSpaceship,
            ((DWORD)(static_cast< IAtlSpaceship*>((_ComMapClass*)8))-8),
            ((_ATL_CREATORARGFUNC*)1)},
            {&IID_IDispatch,
            ((DWORD)(static_cast<IDispatch*>((_ComMapClass*)8))-8),
            ((_ATL_CREATORARGFUNC*)1)},
            {0, 0, 0}
        };
        return _entries;
    }
```

Right now, the `CAtlSpaceship` class supports two interfaces, `IAtlSpaceship` and `IDispatch`, so there are only two entries in the map.
`CComObjectRootEx`'s implementation of `InternalQueryInterface` uses the `_GetEntries()` function as the second parameter. `CComObjectRootEx::InternalQueryInterface` uses a global ATL function named `AtlInternalQueryInterface()` to look up the interface in the map.
`AtlInternalQueryInterface()` simply walks through the map trying to find the interface.
In addition to `COM_INTERFACE_ENTRY`, ATL includes 16 other macros for implementing composition techniques ranging from tear-off interfaces to COM aggregation. Now you'll see what it takes to beef up the `IAtlSpaceship` interface and add those two other interfaces, `IMotion` and `IVisual`. You'll also learn about the strange COM beast known as a dual interface.

Making the Spaceship Go

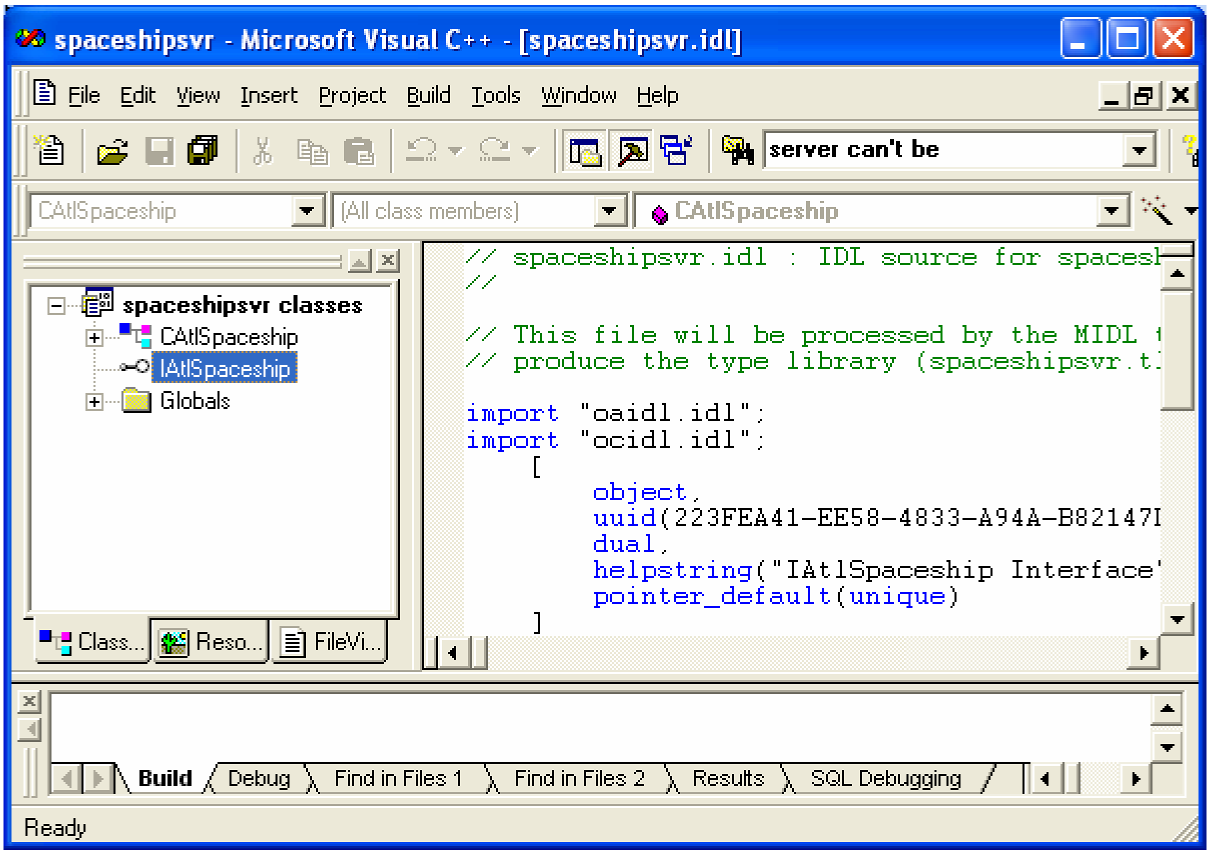
Now that you've got some ATL code staring you in the face, what can you do with it? This is COM, so the place to start is in the **IDL file**. Again, if you're a seasoned C++ developer, this is a new aspect of software development you're probably not used to. Remember that these days, **software distribution** and **integration** are becoming very important. You've been able to get away with hacking out C++ classes and throwing them into a project together because you (as a developer) know the entire picture. However, component technologies (like COM) change that. You as a developer no longer know the entire picture. Often you have only a component, you don't have the source code for the component. The only way to know how to talk to a component is through the **interfaces it exposes**. Keep in mind that modern software developers use many different tools, not just C++. You've got Visual Basic developers, Java developers, Delphi developers, and C developers. COM is all about making the edges line up so that software pieces created by these various components **can all integrate smoothly when they come together**. In addition, distributing software remotely (either out-of-process on the same machine or even to a different machine) requires some sort of inter-process communication. That's why there's **Interface Definition Language** (IDL). Here's the default IDL file created by the ATL wizards with the new spaceship class:

```
// spaceshipsvr.idl : IDL source for spaceshipsvr.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (spaceshipsvr.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";
      [
            object,
            uuid(223FEA41-EE58-4833-A94A-B82147D9B6B4),
            dual,
            helpstring("IAtlSpaceship Interface"),
            pointer_default(unique)
      ]
      interface IAtlSpaceship : IDispatch
      {
      };

[
      uuid(2D1BB358-8D02-45C7-A3FE-3C47B9A1A4AB),
      version(1.0),
      helpstring("spaceshipsvr 1.0 Type Library")
]
library SPACESHIPSVRLib
{
      importlib("stdole32.tlb");
      importlib("stdole2.tlb");

      [
            uuid(334E82E8-1DBF-4E07-8D52-D71CF967FFB1),
            helpstring("AtlSpaceship Class")
      ]
      coclass AtlSpaceship
      {
            [default] interface IAtlSpaceship;
      };
};
```

The key concept involved here is that IDL is a purely **declarative language**. This language defines how other clients will talk to an object. Remember, you'll eventually run this code through the **MIDL compiler** to get a **pure abstract base class** (useful for C++ clients) and a **type library** (useful for Visual Basic and Java clients as well as others). If you understand plain C code, you're well on your way to understanding IDL. You might think of IDL as C with footnotes. The syntax of IDL dictates that attributes will always precede what they describe. For example, attributes precede items such as interface declarations, library declarations, and method parameters.
If you look at the IDL file, you'll notice that it begins by importing **oaidl.idl** and **ocidl.idl**. Importing these files is somewhat akin to including windows.h inside one of your C or C++ files. These IDL files include definitions for all of the basic COM infrastructures (including definitions for `IUnknown` and `IDispatch`).
An open square bracket (**[**) follows the import statement. In IDL, square brackets always **enclose attributes**. The first element described in this IDL file is the `IAtlSpaceship` interface. However, before you can describe the interface, you need to apply some attributes to it. For example, it needs a name (a `GUID`), and you need to tell the MIDL compiler that this interface is COM-oriented rather than being used for standard RPC and that this is a dual interface (more on dual interfaces shortly). Next comes the actual interface itself. Notice how it appears very much like a normal C structure.
Once the interfaces are described in IDL, it is often useful to collect this information into a type library, which is what the next section of the IDL file does. Notice the type library section also begins with an open square bracket, designating that attributes are to follow. As always, the type library is a discrete "thing" in COM and as such requires a name (`GUID`). The library statement tells the MIDL compiler that this library includes a COM class named `AtlSpaceship` and that clients of this class can acquire the `IAtlSpaceship` interface.

Adding Methods to an Interface

Right now the `IAtlSpaceship` interface is pretty sparse. It looks like it could use a method or two. Let's add one. Notice that Visual C++ now extends ClassView to include COM interfaces. You can tell they're COM interfaces because of the little lollipop next to the symbol. Notice also that `CAtlSpaceship` derives from something named `IAtlSpaceship`. `IAtlSpaceship` is, of course, a COM interface. Double-clicking on `IAtlSpaceship` in the ClassView brings that specific section of the IDL into the editor window, as shown in Figure 24.



```
// spaceshipsvr.idl : IDL source for spacesh
//

// This file will be processed by the MIDL t
// produce the type library (spaceshipsvr.tl

import "oaidl.idl";
import "ocidl.idl";
    [
        object,
        uuid(223FEA41-EE58-4833-A94A-B82147I
        dual,
        helpstring("IAtlSpaceship Interface'
        pointer_default(unique)
    ]
```

Figure 24: Interfaces in ClassView.

At this point, you could begin typing the COM interface into the IDL file. If you add functions and methods this way (straight into the IDL file), you'll have to touch the **AtlSpaceship.h** and **AtlSpaceship.cpp** files and insert the methods by hand. A more effective way to add functions to the interface is through the **ClassView**. To edit the IDL through the ClassView, simply right-click the mouse on the interface within ClassView. Two items that appear in the context menu are **Add Method** and **Add Property**. Let's add a method named `CallStarFleet`. Figure 26 shows the dialog box that appears when adding a method.
To add a method, simply type the name of the method, **CallStarFleet** into the **Method Name** text box. Then type the method parameters into the **Parameters** text box as shown below.

```
[in]float fStarDate, [out, retval]BSTR* pbstrRecipient
```

Here's where it helps to understand a little bit about IDL.

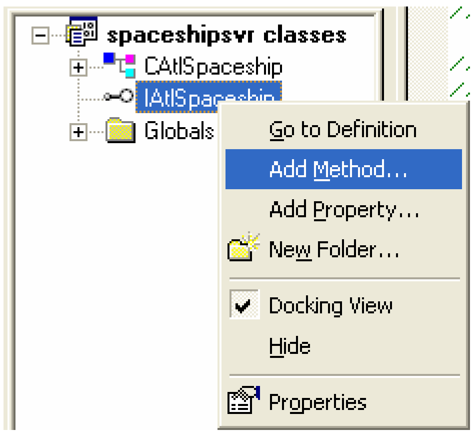
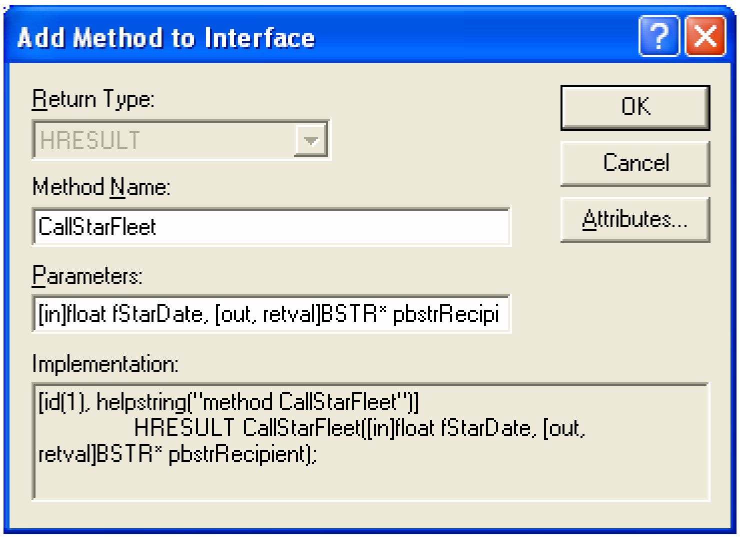


Figure 25: Adding method to interface.



Return Type: HRESULT

Method Name: CallStarFleet

Parameters: [in]float fStarDate, [out, retval]BSTR* pbstrRecipi

Implementation:

```
[id(1), helpstring("method CallStarFleet")]
		HRESULT CallStarFleet([in]float fStarDate, [out,
retval]BSTR* pbstrRecipient);
```

Figure 26: Adding method information.

Remember that IDL's purpose is to provide completely unambiguous information about how methods can be invoked. In the standard C++ world, you could often get away with ambiguities like open-ended arrays because the caller and the callee shared the same stack frame, there was always a lot of wiggle room available. Now that method calls might eventually go over the wire, it's important to tell the remoting layer exactly what to expect when it encounters a COM interface. This is done by applying attributes to the method parameters (more square brackets). The method call shown in Figure 26 (`CallStartFleet()`) has two parameters in its list, a floating point number indicating the **stardate** and a **BSTR** indicating who received the communication. Notice that the method definition spells out the **parameter direction**. The `stardate` is passed **in**to the method call, designated by the `[in]` attribute. A `BSTR` identifying the recipient is passed back as a pointer to a BSTR. The `[out]` attribute indicates the direction of the parameter is from the object back to the client. The `[retval]` attribute indicates that you can assign the result of this method to a variable in higher languages supporting this feature.

Dual Interfaces

If you read through Module 24, you had a chance to see the `IDispatch` interface. `IDispatch` makes it possible to expose functionality (at the binary level) to environments such as VBScript that don't have a clue about **vtables**. For `IDispatch` to work, the client has to go through a lot of machinations before it can call `Invoke()`. The client first has to acquire the invocation tokens. Then the client has to set up the `VARIANT` arguments. On the object side, the object has to decode all those `VARIANT` parameters, make sure they're correct, put them on some sort of stack frame, and then make the function call. As you can imagine, all this work is complex and time-consuming. If you're writing a COM object and you expect some of your clients to use scripting languages and other clients to use languages like C++, you've got a dilemma. You've got to include `IDispatch` or you lock your scripting language clients out. If you provide only `IDispatch`, you make accessing your object from C++ very inconvenient. Of course, you can provide access through both IDispatch and a custom interface, but that involves a lot of bookkeeping work. Dual interfaces evolved to handle this problem.
A dual interface is simply `IDispatch` with functions pasted onto the end. For example, the `IMotion` interface described below is a valid dual interface:

```
interface IMotion : public IDispatch
{
    virtual HRESULT Fly() = 0;
    virtual HRESULT GetPosition() = 0;
};
```

Because `IMotion` derives from `IDispatch`, the first seven functions of `IMotion` are those of `IDispatch`. Clients who understand only `IDispatch` (VBScript for instance) look at the interface as just another version of `IDispatch` and feed `DISPID`s to the Invoke function in the hopes of invoking a function. Clients who understand vtable-style custom interfaces look at the entire interface, ignore the middle four functions (the `IDispatch`

functions), and concentrate on the first three functions (IUnknown) and the last three functions (the ones that represent the interface's core functions). Figure 27 shows the vtable layout of IMotion.
Most raw C++ implementations load the type library right away and delegate to ITypeInfo to perform the nasty task of implementing Invoke() and GetIDsOfNames(). To get an idea of how this works, see Kraig Brockschmidt's book Inside OLE, 2d. ed. (Microsoft Press, 1995) or Dale Rogerson's book Inside COM (Microsoft Press, 1997).

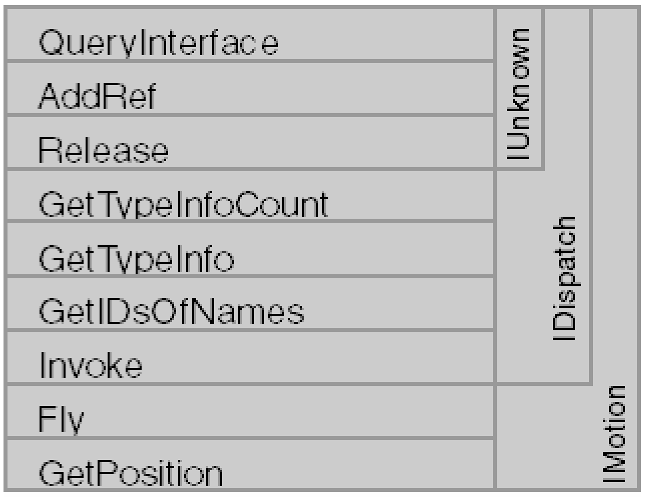


Figure 27: The layout of a dual interface.

ATL and IDispatch

ATL's implementation of IDispatch delegates to the type library. ATL's implementation of IDispatch lives in the class IDispatchImpl. Objects that want to implement a dual interface include the IDispatchImpl template in the inheritance list like this:

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IVisual, &IID_IVisual,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
                         &LIBID_SPACESHIPSVRLib>
{...};
```

In addition to including the IDispatchImpl template class in the inheritance list, the object includes entries for the dual interface and for IDispatch in the interface map so that QueryInterface() works properly:

```
BEGIN_COM_MAP(CAtlSpaceship)
    COM_INTERFACE_ENTRY(IAtlSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

As you can see, the IDispatchImpl template class arguments include the dual interface itself, the GUID for the interface, and the GUID representing the type library holding all the information about the interface. In addition to these template arguments, the IDispatchImpl class has some optional parameters not illustrated in Figure 27. The template parameter list also includes room for a major and minor version of the type library. Finally, the last template parameter is a class for managing the type information. ATL provides a default class named CComTypeInfoHolder.
In most raw C++ implementations of IDispatch, the class calls LoadTypeLib() and ITypeLib::GetTypeInfoOfGuid in the constructor and holds on to the ITypeInfo pointer for the life of

the class. ATL's implementation does things a little differently by using the CComTypeInfoHolder class to help manage the ITypeInfo pointer. CComTypeInfoHolder maintains an ITypeInfo pointer as a data member and wraps the critical IDispatch-related functions GetIDsOfNames() and Invoke().
Clients acquire the dual interface by calling QueryInterface() for IID_IAtlSpaceship. The client can also get this interface by calling QueryInterface() for IDispatch. If the client calls CallStartFleet() on the interface, the client accesses those functions directly (as for any other COM interface).
When a client calls IDispatch::Invoke, the call lands inside IDispatchImpl's Invoke() function as you'd expect. From there, IDispatchImpl::Invoke delegates to the CComTypeInfoHolder class to perform the invocation, CComTypeInfoHolder's Invoke(). The CComTypeInfoHolder class doesn't call LoadTypeLib() until an actual call to Invoke() or GetIDsOfNames(). CComTypeInfoHolder has a member function named GetTI() that consults the Registry for the type information (using the GUID and any major/minor version numbers passed in as a template parameter). Then CComTypeInfoHolder calls ITypeLib::GetTypeInfo to get the information about the interface. At that point, the type information holder delegates to the type information pointer. IDispatchImpl implements IDispatch::GetIDsOfNames in the same manner.

The IMotion and IVisual Interfaces

To get this COM class up to snuff with the other versions (the raw C++ version and the MFC version described in Module 23), you need to add the IMotion and IVisible interfaces to the project and to the class. Unfortunately, at the present time the only way to get this to happen is by typing the interfaces in by hand (the ATL AppWizard gives you only one interface by default but Visual /Studio C++ .Net has improved this). Open the IDL file and position the cursor near the top (somewhere after the #import statements but before the library statement), and start typing interface definitions as described in the following paragraph.
Once you get the hang of IDL, your first instinct when describing an interface should be to insert an open square bracket. Remember that in IDL, distinct items get attributes. One of the most important attributes for an interface is the name, or the GUID. In addition, at the very least the interface has to have the object attribute to tell the MIDL compiler you're dealing with COM at this point (as opposed to regular RPC). You also want these interfaces to be dual interfaces. The keyword "dual" in the interface attributes indicates this and inserts certain Registry entries to get the universal marshaling working correctly. After the attributes are closed off with a closing square bracket, the interface keyword kicks in to describe the interface. You'll make IMotion a dual interface and IVisual a plain custom interface to illustrate how the two different types of interfaces are attached to the CSpaceship class. Here are the IMotion and IVisible interfaces described in IDL:

```
[
    object,
    uuid(97B5C101-5299-11d1-8CAA-FD10872CC837),
    dual,
    helpstring("IMotion interface")
]

interface IMotion : IDispatch
{
    HRESULT Fly();
    HRESULT GetPosition([out,retval]long* nPosition);
};

[
    object,
    uuid(56F58464-52A4-11d1-8CAA-FD10872CC837),
    helpstring("IVisual interface")
]

interface IVisual : IUnknown
{
    HRESULT Display();
};
```

Once the interfaces are described in IDL, you run the IDL through the MIDL compiler again. The MIDL compiler spits out a new copy of **spaceshipsvr.h** with the pure abstract base classes for IMotion and IVisual.
Now you need to add these interfaces to the CSpaceship class. There are two steps here. The first step is to create the interface part of the COM class's identity. Let's do the IMotion interface first. Adding the IMotion interface to CSpaceship is easy. Just use the IDispatchImpl template to provide an implementation of a dual interface like this:

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
                         &LIBID_SPACESHIPSVRLib>
{...};
```

The second step involves beefing up the interface map so the client can acquire the IMotion interface. However, having two dual interfaces in a single COM class brings up an interesting issue. When a client calls QueryInterface() for IMotion, the client should definitely get IMotion. However, when the client calls QueryInterface() for IDispatch, which version of IDispatch should the client get, IAtlSpaceship's dispatch interface or IMotion's dispatch interface?

Multiple Dual Interfaces

Remember that all dual interfaces begin with the seven functions of IDispatch. A problem occurs whenever the client calls QueryInterface() for IID_IDispatch. As a developer, you need to choose which version of IDispatch to pass out. The interface map is where the QueryInterface() for IID_IDispatch is specified. ATL has a specific macro for handling the dual interface situation. First consider the interface map for CAtlSpaceship so far:

```
BEGIN_COM_MAP(CAtlSpaceship)
    COM_INTERFACE_ENTRY(IAtlSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

When the client calls QueryInterface(), ATL rips through the table trying to match the requested IID to one in the table. The interface map above handles two interfaces: IAtlSpaceship and IDispatch. If you want to add another dual interface to the CAtlSpaceship class, you need a different macro.
The macro handling multiple dispatch interfaces in an ATL-based COM class is named COM_INTERFACE_ENTRY2. To get QueryInterface() working correctly, all you need to do is decide which version of IDispatch the client should get when asking for IDispatch, like this:

```
BEGIN_COM_MAP(CAtlSpaceship)
    COM_INTERFACE_ENTRY(IAtlSpaceship)
    COM_INTERFACE_ENTRY(IMotion)
    COM_INTERFACE_ENTRY2(IDispatch, IAtlSpaceship)
END_COM_MAP()
```

In this case, a client asking for IDispatch gets a pointer to IAtlSpaceship (whose first seven functions include the IDispatch functions). Adding a nondual interface to an ATL-based COM class is even easier. Just add the interface to the inheritance list like this:

```
class ATL_NO_VTABLE CAtlSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAtlSpaceship, &CLSID_AtlSpaceship>,
    public IDispatchImpl<IAtlSpaceship, &IID_IAtlSpaceship,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
                         &LIBID_SPACESHIPSVRLib>,
    public IVisual
{...};
```

Then add an interface map entry like this:

```
BEGIN_COM_MAP(CAtlSpaceship)
    COM_INTERFACE_ENTRY(IAtlSpaceship)
    COM_INTERFACE_ENTRY(IMotion)
    COM_INTERFACE_ENTRY2(IDispatch, IAtlSpaceship)
END_COM_MAP()
```

Then, repeat for `IVisual` interface.

Conclusion

There are a couple of key points in this module to remember. **COM is a binary object model**. Clients and objects agree on a binary layout (the interface). Once both parties agree on the layout, they talk together via the interface. The client is not at all concerned about how that interface is actually wired up. As long as the functions work as advertised, the client is happy. There are a number of ways to hook up COM interfaces, including multiply inheriting a single C++ class from several interfaces, using nested classes, or using a framework such as ATL.
ATL is Microsoft's framework for assembling small COM classes. ATL has two sides, some smart pointers to help with client-side coding and a complete framework for implementing COM classes. ATL implements `IUnknown`, `IDispatch`, and `IClassFactory` in templates provided through the library. In addition, ATL includes a wizard for helping you get started with a COM server and a wizard for inserting COM classes into your project.
While ATL does a lot for you, it doesn't completely excuse you from learning the basics of how COM works. In fact, you'll be able to use ATL a lot more efficiently once you understand COM. In the next module, we'll take a look at how to use ATL to write ActiveX controls effectively.

`CComPtr` class info:

Information	Description
The class	`CComPtr`.
The use	A smart pointer class for managing COM interface pointers.

The prototype	`template<class T>`
	`class CComPtr`

The parameters	`T` A COM interface specifying the type of pointer to be stored.
Method	`CComPtr` - The constructor.
Operator	`operator =` - Assigns a pointer to the member pointer.
The header file	`atlcomcli.h`
Remark	ATL uses `CComPtr` and `CComQIPtr` to manage COM interface pointers. Both are derived from `CComPtrBase`, and both perform automatic reference counting.

Table 1.

`CComQIPtr` class info:

Information	Description
The class	`CComQIPtr`.
The use	A smart pointer class for managing COM interface pointers.

The prototype	`template<`
	`   class T,`
	`   const IID* piid = &__uuidof(T)`
	`>`
	`class CComQIPtr: public CComPtr<T>`

The parameters	`T` A COM interface specifying the type of pointer to be stored.

	`piid` A pointer to the IID of T.
Method	`CComQIPtr` - Constructor.
Operator	`operator =` - Assigns a pointer to the member pointer.
The header file	`atlcomcli.h`
Remark	ATL uses `CComQIPtr` and `CComPtr` to manage COM interface pointers, both of which derive from `CComPtrBase`. Both classes perform automatic reference counting through calls to `AddRef()` and `Release()`. Overloaded operators handle pointer operations.

Table 1.

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

Further reading and digging:

1. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
2. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
3. MSDN Library
4. DCOM at MSDN.
5. COM+ at MSDN.
6. COM at MSDN.
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.