**Unit-3**

**Chapter- 24:**

**The Component Object Model**

**Q: Describe the COM? What are the problems that COM solves?**

**Ans:**

# The Component Object Model

COM is an "industry-standard" software architecture supported by Microsoft, Digital Equipment Corporation, and many other companies. It's by no means the only standard. Indeed, it competes directly against other standards, such as Corba from the Open Software Foundation (OSF). Some people are working to establish interoperability between COM and other architectures.

## The Essence of COM

COM provides a unified, expandable, object-oriented communications protocol for Windows that already supports the following features:

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

COM is a powerful integrating technology that allows you to mix all sorts of disparate software parts together at runtime. COM allows developers to write software that runs together regardless of issues such as thread-awareness and language choice.

COM is a protocol that connects one software module with another and then drops out of the picture. After the connection is made, the two modules can communicate through a mechanism called an interface. Interfaces require no statically or dynamically linked entry points or hard-coded addresses other than the few general-purpose COM functions that start the communication process. An interface (more precisely, a COM interface) is a term that you'll be seeing a lot of.

**Q: Describe the COM? Explain COM Interface?**

**Ans:**

Before digging into the topic of interfaces, let's re-examine the nature of inheritance and polymorphism in normal C++. We'll use a planetary-motion simulation (suitable for NASA or Nintendo) to illustrate C++ inheritance and polymorphism. Imagine a spaceship that travels through our solar system under the influence of the sun's gravity. In ordinary C++, you could declare a *CSpaceship* class and write a constructor that sets the spaceship's initial position and acceleration. Then you could write a nonvirtual member function named *Fly* that implemented Kepler's laws to model the movement of the spaceship from one position to the next—say, over a period of 0.1 second. You could also write a *Display* function that painted an image of the spaceship in a window. The most interesting feature of the *CSpaceship* class is that the interface of the C++ class (the way the client talks to the class) and the implementation are tightly bound. One of the main goals of COM is to separate a class's interface from its implementation.

If we think of this example within the context of COM, the spaceship code could exist as a separate EXE or DLL (the component), which is a COM module. In COM the simulation manager (the client program) can't call *Fly* or any *CSpaceship* constructor directly: COM provides only a standard global function to gain access to the spaceship object, and then the client and the object use interfaces to talk to one another. Before we tackle real COM, let's build a COM simulation in which both the component and the client code are statically linked in the same EXE file. For our standard global function, we'll invent a function named *GetClassObject.*

In this COM simulation, clients will use this global single abstract function (*GetClassObject*) for objects of a particular class. In real COM, clients would get a class object first and then ask the class object to manufacture the real object in much the same way MFC does dynamic creation. *GetClassObject* has the following three parameters:

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

The first *GetClassObject* parameter, *nClsid*, is a 32-bit integer that uniquely identifies the *CSpaceship* class. The second parameter, *nIid*, is the unique identifier of the interface that we want. The third parameter is a pointer to an interface to the object. Remember that we're going to be dealing with interfaces now, (which are different from classes). As it turns out, a class can have several interfaces, so the last two parameters exist to manage interface selection. The function returns *TRUE* if the call is successful.

Now let's back up to the design of *CSpaceship*. We haven't really explained spaceship interfaces yet. A COM interface is a C++ base class (actually, a C++ *struct*) that declares a group of pure virtual functions. These functions completely control some aspect of derived class behavior. For *CSpaceship*, let's write an interface named *IMotion*, which controls the spaceship object's position. For simplicity's sake, we'll declare just two functions, *Fly* and *GetPosition*, and we'll keep things uncomplicated by making the position value an integer. The *Fly* function calculates the position of the spaceship, and the *GetPosition* function returns a reference to the current position. Here are the declarations:

struct IMotion

{

virtual void Fly() = 0;

virtual int& GetPosition() = 0;

};

class CSpaceship : public IMotion

{

protected:

int m\_nPosition;

public:

CSpaceship() { m\_nPosition = 0; }

void Fly();

int& GetPosition() { return m\_nPosition; }

};

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

IMotion\* pMot;

GetClassObject(CLSID\_CSpaceship, IID\_IMotion, (void\*\*) &pMot);

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

int nPos = 50;

pMot->GetPosition() = nPos;

pMot->Fly();

nPos = pMot->GetPosition();

TRACE("new position = %d\n", nPos);

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

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

struct IVisual

{

virtual void Display() = 0;

};

Well, in your space simulation, you probably want to include other kinds of objects in addition to 0spaceships. Imagine that the *IMotion* and *IVisual* interfaces are being used for other classes. Perhaps a *CSun* class has an implementation of *IVisual* but does not have an implementation of *IMotion*, and perhaps a *CSpaceStation* class has other interfaces as well. If you "published" your *IMotion* and *IVisual* interfaces, perhaps other space simulation software companies would adopt them.

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

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

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

class CSpaceship

{

protected:

int m\_nPosition;

int m\_nAcceleration;

int m\_nColor;

public:

CSpaceship()

{ m\_nPosition = m\_nAcceleration = m\_nColor = 0; }

class XMotion : public IMotion

{

public:

XMotion() { }

virtual void Fly();

virtual int& GetPosition();

} m\_xMotion;

class XVisual : public IVisual

{

public:

XVisual() { }

virtual void Display();

} m\_xVisual;

friend class XVisual;

friend class XMotion;

};

Notice that the implementations of *IMotion* and *IVisual* are contained within the "parent" *CSpaceship* class. In COM, this parent class is known as the class with object identity. Be aware that *m\_xMotion* and *m\_xVisual* are actually embedded data members of *CSpaceship.* Indeed, you could have implemented *CSpaceship* strictly with embedding. Nesting, however, brings to the party two advantages : 1) nested class member functions can access parent class data members without the need for *CSpaceship* pointer data members, and 2) the nested classes are neatly packaged along with the parent while remaining invisible outside the parent. Look at the code below for the *GetPosition* member function.

int& CSpaceship::XMotion::GetPosition()

{

METHOD\_PROLOGUE(CSpaceship, Motion) // makes pThis

return pThis->m\_nPosition;

}

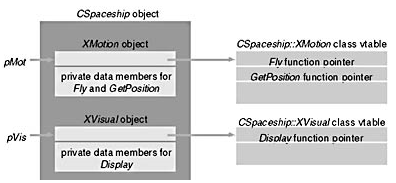
Notice also the double scope resolution operators, which are necessary for nested class member functions. *METHOD\_PROLOGUE* is a one-line MFC macro that uses the C *offsetof* operator to retrieve the offset used in generating a *this* pointer to the parent class, *pThis*. The compiler always knows the offset from the beginning of parent class data to the beginning of nested class data. *GetPosition* can thus access the *CSpaceship* data member *m\_nPosition*.

Now suppose you have two interface pointers, *pMot* and *pVis*, for a particular *CSpaceship* object. (Don't worry yet about how you got these pointers.) You can call interface member functions in the following manner:

pMot->Fly();

pVis->Display();

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



## The *IUnknown* Interface and the *QueryInterface* Member Function

COM declares a special interface named *IUnknown* for this purpose. As a matter of fact, all interfaces are derived from *IUnknown*, which has a pure virtual member function, *QueryInterface*, that returns an interface pointer based on the interface ID you feed it.

Once the interface mechanisms are hooked up, the client needs to get an *IUnknown* interface pointer (at the very least) or a pointer to one of the derived interfaces. Here is the new interface hierarchy, with *IUnknown* at the top:

struct IUnknown

{

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

};

struct IMotion : public IUnknown

{

virtual void Fly() = 0;

virtual int& GetPosition() = 0;

};

struct IVisual : public IUnknown

{

virtual void Display() = 0;

};

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



*GetClassObject* can get the interface pointer for a given *CSpaceship* object by getting the address of the corresponding embedded object. Here's the code for the *QueryInterface* functionin *XMotion*:

BOOL CSpaceship::XMotion::QueryInterface(int nIid,

void\*\* ppvObj)

{

METHOD\_PROLOGUE(CSpaceship, Motion)

switch (nIid) {

case IID\_IUnknown:

case IID\_IMotion:

\*ppvObj = &pThis->m\_xMotion;

break;

case IID\_IVisual:

\*ppvObj = &pThis->m\_xVisual;

break;

default:

\*ppvObj = NULL;

return FALSE;

}

return TRUE;

}

Because *IMotion* is derived from *IUnknown*, an *IMotion* pointer is a valid pointer if the caller asks for an *IUnknown* pointer.

Below is a *GetClassObject* function that usesthe address of *m\_xMotion* to obtain the first interface pointer for the newly constructed *CSpaceship* object:

BOOL GetClassObject(int& nClsid, int& nIid,

void\*\* ppvObj)

{

ASSERT(nClsid == CLSID\_CSpaceship);

CSpaceship\* pObj = new CSpaceship();

IUnknown\* pUnk = &pObj->m\_xMotion;

return pUnk->QueryInterface(nIid, ppvObj);

}

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

IMotion\* pMot;

IVisual\* pVis;

GetClassObject(CLSID\_CSpaceship, IID\_IMotion, (void\*\*) &pMot);

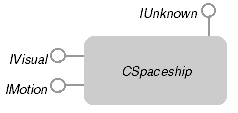
pMot->Fly();

pMot->QueryInterface(IID\_IVisual, (void\*\*) &pVis);

pVis->Display();

Notice that the client uses a *CSpaceship* object, but it never has an actual *CSpaceship* pointer. Thus, the client cannot directly access *CSpaceship* data members even if they're public. Notice also that we haven't tried to delete the spaceship object yet—that will come shortly.

There's a special graphical representation for interfaces and COM classes. Interfaces are shown as small circles (or jacks) with lines attached to their class. The *IUnknown* interface, which every COM class supports, is at the top, and the others are on the left. The *CSpaceship* class can be represented like this.



## Reference Counting: The *AddRef* and *Release* Functions

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

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

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

DWORD CSpaceship::XMotion::Release()

{

METHOD\_PROLOGUE(CSpaceship, Motion) // makes pThis

if (pThis->m\_dwRef == 0)

return 0;

if (--pThis->m\_dwRef == 0) {

delete pThis; // the spaceship object

return 0;

}

return pThis->m\_dwRef;

}

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

**Q: What is Class Factories?**

**Ans:**

## Class Factories:

The COM literature often uses the term "component object" to refer to the object plus the code associated with it. COM carries with it the notion of a "class object," which is sometimes referred to as a "class factory." To be more accurate, it should probably be called an "object factory." A COM class object represents the global static area of a specific COM class. Its analog in MFC is the *CRuntimeClass*. A class object is sometimes called a classfactory because it often implements a special COM interface named *IClassFactory*. This interface, like all interfaces, is derived from *IUnknown*. *IClassFactory*'s principal member function is *CreateInstance*, which in our COM simulation is declared like this:

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

**Why use a class factory?**

The component provides the class factory for this purpose and thus encapsulates the creation step, as it should. Locating and launching component modules—and thus establishing the class factory—is expensive, but constructing objects with *CreateInstance* is cheap. We can therefore allow a single class factory to create multiple objects.

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

**Chapter-25:**

**Automation**

**Q: Write short note on Automation?**

**Ans:**

# Automation Clients and Automation Components

A clearly defined "master-slave" relationship is always present in an Automation communication dialog. The master is the Automation client and the slave is the Automation component (server). The client initiates the interaction by constructing a component object (it might have to load the component program) or by attaching to an existing object in a component program that is already running. The client then calls interface functions in the component and releases those interfaces when it's finished.

Here are some interaction scenarios:

* A C++ Automation client uses a Microsoft or third-party application as a component. The interaction could trigger the execution of VBA code in the component application.
* A C++ Automation component is used from inside a Microsoft application (or a Visual Basic application), which acts as the Automation client. Thus, VBA code can construct and use C++ objects.
* A C++ Automation client uses a C++ Automation component.
* A Visual Basic program uses an Automation-aware application such as Excel. In this case, Visual Basic is the client and Excel is the component.

# Properties, Methods, and Collections

The distinction between a property and a method is somewhat artificial. Basically, a property is a value that can be both set and retrieved. You can, for example, set and get the Selection property for an Excel application. Another example is Excel's Width property, which applies to many object types. Some Excel properties are read-only; most are read/write.

Properties don't officially have parameters, but some properties are indexed. The property index acts a lot like a parameter. It doesn't have to be an integer, and it can have more than one element (row and column, for example). You'll find many indexed properties in Excel's object model, and Excel VBA can handle indexed properties in Automation components.

Methods are more flexible than properties. They can have zero or many parameters, and they can either set or retrieve object data. Most frequently they perform some action, such as showing a window. Excel's *Select* method is an example of an action method.

The Excel object model supports collection objects. If you use the Worksheets property of the *Application* object, you get back a *Sheets* collection object, which represents all the worksheets in the active workbook. You can use the Item property (with an integer index) to get a specific *Worksheet* object from a Sheets collection, or you can use an integer index directly on the collection.

# The Problem That Automation Solves

Automation's general-purpose interface, *IDispatch*, serves the needs of both C++ and VBA programmers. As you might guess from your glimpse of Excel VBA, this interface involves objects, methods, and properties.

You can write COM interfaces that include functions with any parameter types and return values you specify. You can solve the communication problem with one interface that has a member function smart enough to accommodate methods and properties as defined by VBA. Needless to say, *IDispatch* has such a function: *Invoke*. You use *IDispatch::Invoke* for COM objects that can be constructed and used in either C++ or VBA programs.

It funnels all intermodule communication through the *IDispatch::Invoke* function. How does a client first connect to its component? Because *IDispatch* is merely another COM interface, all the registration logic supported by COM comes into play. Automation components can be DLLs or EXEs, and they can be accessed over a network using distributed COM (DCOM).

# The *IDispatch* Interface

*IDispatch* is the heart of Automation. It's fully supported by COM marshaling, as are all the other standard COM interfaces, and it's supported well by the MFC library. At the component end, you need a COM class with an *IDispatch* interface (plus the prerequisite class factory, of course). At the client end, you use standard COM techniques to obtain an *IDispatch* pointer.

Remember that *Invoke* is the principal member function of *IDispatch*. If you looked up *IDispatch::Invoke* in the Visual C++ online documentation, you'd see a really ugly set of parameters. Don't worry about those now. The MFC library steps in on both sides of the *Invoke* call, using a data-driven scheme to call component functions based on dispatch map parameters that you define with macros.

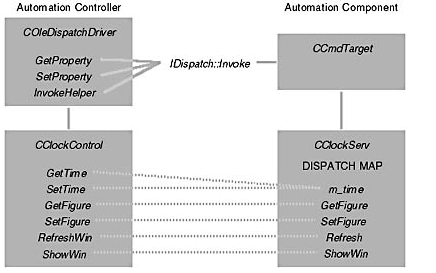
*Invoke* isn't the only *IDispatch* member function. Another function your controller might call is *GetIDsOfNames*. From the VBA programmer's point of view, properties and methods have symbolic names, but C++ programmers prefer more efficient integer indexes. *Invoke* uses integers to specify properties and methods, so *GetIDsOfNames* is useful at the start of a program for converting each name to a number if you don't know the index numbers at compile time. You've already seen that *IDispatch* supports symbolic names for methods. In addition, the interface supports symbolic names for a method's parameters. The *GetIDsOfNames* function returns those parameter names along with the method name. Unfortunately, the MFC *IDispatch* implementation doesn't support named parameters.

# An MFC Automation Client Program

The MFC library provides a base class *COleDispatchDriver* for this purpose. This class has a data member, *m\_lpDispatch*, which contains the corresponding component's *IDispatch* pointer. To shield you from the complexities of the *Invoke* parameter sequence, *COleDispatchDriver* has several member functions, including *InvokeHelper*, *GetProperty*, and *SetProperty*. These three functions call *Invoke* for an *IDispatch* pointer that links to the component. The *COleDispatchDriver* object incorporates the *IDispatch* pointer.

The function parameters identify the property or method, its return value, and its parameters. You'll learn about dispatch function parameters later, but for now take special note of the first parameter for the *InvokeHelper*, *GetProperty*, and *SetProperty* functions. This is the unique integer index, or dispatch ID (DISPID), for the property or method. Because you're using compiled C++, you can establish these IDs at compile time. If you're using an MFC Automation component with a dispatch map, the indexes are determined by the map sequence, beginning with 1. If you don't know a component's dispatch indexes, you can call the *IDispatch* member function *GetIDsOfNames* to convert the symbolic property or method names to integers.

The following illustration shows the interactions between the client (or controller) and the component.



The solid lines show the actual connections through the MFC base classes and the *Invoke* function. The dotted lines represent the resulting logical connections between client class members and component class members.

**Chapter-26:**

**Uniform Data Transfer—Clipboard Transfer and OLE Drag and Drop**

**Q: Write short note on UDT and Chipboard Transfer?**

**Ans:**

ActiveX technology includes a powerful mechanism for transferring data within and among Microsoft Windows-based applications. The COM *IDataObject* interface is the key element of what is known as UniformDataTransfer. As you'll see, Uniform Data Transfer (UDT) gives you all sorts of options for the formatting and storage of your transferred data, going well beyond standard clipboard transfers.

# MFC Uniform Data Transfer Support

The MFC library does a lot to make data object programming easier. As you study the MFC data object classes, you'll start to see a pattern in MFC COM support. At the component end, the MFC library provides a base class that implements one or more OLE interfaces. The interface member functions call virtual functions that you override in your derived class. At the client end, the MFC library provides a class that wraps an interface pointer. You call simple member functions that use the interface pointer to make COM calls.

The terminology needs some clarification here. The data object that's been described is the actual C++ object that you construct, and that's the way Brockschmidt uses the term. In the MFC documentation, a data object is what the client program sees through an *IDataObject* pointer. A datasource is the object you construct in a component program.

## The *COleDataSource* Class

When you want to use a data source, you construct an object of class *COleDataSource*, which implements the *IDataObject* interface (without advisory connection support). This class builds and manages a collection of data formats stored in a cache in memory. A data source is a regular COM object that keeps a reference count. Usually, you construct and fill a data source, and then you pass it to the clipboard or drag and drop it in another location, never to worry about it again. If you decide not to pass off a data source, you can invoke the destructor, which cleans up all its formats.

Following are some of the more useful member functions of the *COleDataSource* class.

### void CacheData(CLIPFORMAT *cfFormat*, STGMEDIUM\* *lpStgMedium*, FORMATETC\* *lpFormatEtc* = NULL);

This function inserts an element in the data object's cache for data transfer. The *lpStgMedium* parameter points to the data, and the *lpFormatEtc* parameter describes the data. If, for example, the *STGMEDIUM* structure specifies a disk filename, that filename gets stored inside the data object. If *lpFormatEtc* is set to *NULL*, the function fills in a *FORMATETC* structure with default values. It's safer, though, if you create your *FORMATETC* variable with the *tymed* member set.

### void CacheGlobalData(CLIPFORMAT *cfFormat*, HGLOBAL *hGlobal*, FORMATETC\* *lpFormatEtc* = NULL);

You call this specialized version of *CacheData* to pass data in global memory (identified by an *HGLOBAL* variable). The data source object is considered the owner of that global memory block, so you should not free it after you cache it. You can usually omit the *lpFormatEtc* parameter. The *CacheGlobalData* function does not make a copy of the data.

### DROPEFFECT DoDragDrop(DWORD *dwEffects* = DROPEFFECT\_COPY|DROPEFFECT\_MOVE| DROPEFFECT\_LINK, LPCRECT *lpRectStartDrag* = NULL, COleDropSource\* *pDropSource* = NULL);

You call this function for drag-and-drop operations on a data source.

### void SetClipboard(void);

The *SetClipboard* function, which you'll see in the EX26A example, calls the *OleSetClipboard* function to put a data source on the Windows Clipboard. The clipboard is responsible for deleting the data source and thus for freeing the global memory associated with the formats in the cache. When you construct a *COleDataSource* object and call *SetClipboard*, COM calls *AddRef* on the object.

## The *COleDataObject* Class

This class is on the destination side of a data object transfer. Its base class is *CCmdTarget*, and it has a public member *m\_lpDataObject* that holds an *IDataObject* pointer. That member must be set before you can effectively use the object. The class destructor only calls *Release* on the *IDataObject* pointer.

Following are a few of the more useful *COleDataObject* member functions.

### BOOL AttachClipboard(void);

As Brockschmidt points out, OLE clipboard processing is internally complex. From your point of view, however, it's straightforward—as long as you use the *COleDataObject* member functions. You first construct an "empty" *COleDataObject* object, and then you call *AttachClipboard*, which calls the global *OleGetClipboard* function. Now the *m\_lpDataObject* data member points back to the source data object (or so it appears), and you can access its formats.

If you call the *GetData* member function to get a format, you must remember that the clipboard owns the format and you cannot alter its contents. If the format consists of an *HGLOBAL* pointer, you must not free that memory and you cannot hang on to the pointer. If you need to have long-term access to the data in global memory, consider calling *GetGlobalData* instead.

If a non-COM-aware program copies data onto the clipboard, the *AttachClipboard* function still works because COM invents a data object that contains formats corresponding to the regular Windows data on the clipboard.

### void BeginEnumFormats(void); BOOL GetNextFormat(FORMATETC\* *lpFormatEtc*);

These two functions allow you to iterate through the formats that the data object contains. You call *BeginEnumFormats* first, and then you call *GetNextFormat* in a loop until it returns *FALSE*.

### BOOL GetData(CLIPFORMAT *cfFormat*, STGMEDIUM\* *lpStgMedium* FORMATETC\* *lpFormatEtc* = NULL);

This function calls *IDataObject::GetData* and not much more. The function returns *TRUE* if the data source contains the format you asked for. You generally need to supply the *lpFormatEtc* parameter.

### HGLOBAL GetGlobalData(CLIPFORMAT *cfFormat*, FORMATETC\* *lpFormatEtc* = NULL);

Use the *GetGlobalData* function if you know your requested format is compatible with global memory. This function makes a copy of the selected format's memory block, and it gives you an *HGLOBAL* handle that you must free later. You can often omit the *lpFormatEtc* parameter.

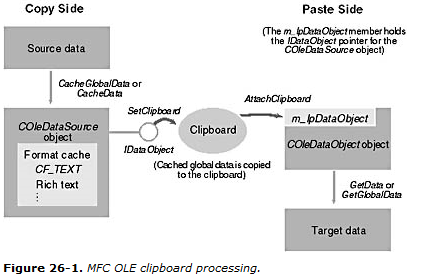
### BOOL IsDataAvailable(CLIPFORMAT *cfFormat*, FORMATETC\* *lpFormatEtc* = NULL);

The *IsDataAvailable* function tests whether the data object contains a given format.

## MFC Data Object Clipboard Transfer

Now that you've seen the *COleDataObject* and *COleDataSource* classes, you'll have an easy time doing clipboard data object transfers. But why not just do clipboard transfers the old way with *GetClipboardData* and *SetClipboardData*? You could for most common formats, but if you write functions that process data objects, you can use those same functions for drag and drop.

Figure 26-1 shows the relationship between the clipboard and the *COleDataSource* and *COleDataObject* classes. You construct a *COleDataSource* object



on the copy side, and then you fill its cache with formats. When you call *SetClipboard*, the formats are copied to the clipboard. On the paste side, you call *AttachClipboard* to attach an *IDataObject* pointer to a *COleDataObject* object, after which you can retrieve individual formats.

Suppose you have a document-view application whose document has a *CString* data member *m\_strText*. You want to use view class command handler functions that copy to and paste from the clipboard. Before you write those functions, write two helper functions. The first, *SaveText*, creates a data source object from the contents of *m\_strText*. The function constructs a *COleDataSource* object, and then it copies the string contents to global memory. Last it calls *CacheGlobalData* to store the *HGLOBAL* handle in the data source object.

# MFC Drag and Drop

Drag and drop was the ultimate justification for the data object code you've been looking at. OLE supports this feature with its *IDropSource* and *IDropTarget* interfaces plus some library code that manages the drag-and-drop process. The MFC library offers good drag-and-drop support at the view level, so we'll use it. Be aware that drag-and-drop transfers are immediate and independent of the clipboard. If the user cancels the operation, there's no "memory" of the object being dragged.

Drag-and-drop transfers should work consistently between applications, between windows of the same application, and within a window. When the user starts the operation, the cursor should change to an arrow\_rectangle combination. If the user holds down the Ctrl key, the cursor turns into a plus sign (+), which indicates that the object is being copied rather than moved.

MFC also supports drag-and-drop operations for items in compound documents. This is the next level up in MFC OLE support, and it's not covered in this chapter. Look up the OCLIENT example in the online documentation under Visual C++ Samples.

## The Source Side of the Transfer

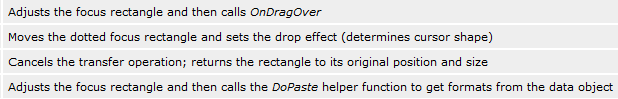
When your source program starts a drag-and-drop operation for a data object, it calls *COleDataSource::DoDragDrop*. This function internally creates an object of MFC class *COleDropSource*, which implements the *IOleDropSource* interface. *DoDragDrop* is one of those functions that don't return for a while. It returns when the user drops the object or cancels the operation or when a specified number of milliseconds have elapsed.

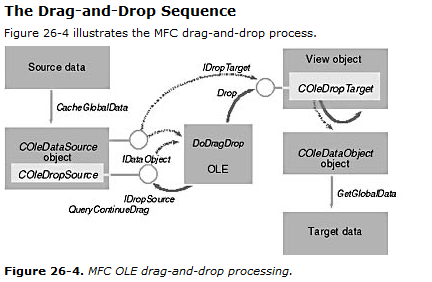
If you're programming drag-and-drop operations to work with a *CRectTracker* object, you should call *DoDragDrop* only when the user clicks inside the tracking rectangle, not on its border. *CRectTracker::HitTest* gives you that information. When you call *DoDragDrop*, you need to set a flag that tells you whether the user is dropping the object into the same view (or document) that it was dragged from.

## The Destination Side of the Transfer

If you want to use the MFC library's view class drag-and-drop support, you must add a data member of class *COleDropTarget* to your derived view class. This class implements the *IDropTarget* interface, and it holds an *IDropSource* pointer that links back to the *COleDropSource* object. In your view's *OnInitialUpdate* function, you call the *Register* member function for the embedded *COleDropTarget* object.

After you have made your view a drop target, you must override four *CView* virtual functions, which the framework calls during the drag-and-drop operation. Here's a summary of what they should do, assuming that you're using a tracker.





Here's a summary of what's going on:

1. User presses the left mouse button in the source view window.
2. Mouse button handler calls *CRectTracker::HitTest* and finds out that the cursor was inside the tracker rectangle.
3. Handler stores formats in a *COleDataSource* object.
4. Handler calls *COleDataSource::DoDragDrop* for the data source.
5. User moves the cursor to the view window of the target application.
6. OLE calls *IDropTarget::OnDragEnter* and *OnDragOver* for the *COleDropTarget* object, which calls the corresponding virtual functions in the target's view. The *OnDragOver* function is passed a *COleDataObject* pointer for the source object, which the target tests for a format it can understand.
7. *OnDragOver* returns a drop effect code, which OLE uses to set the cursor.
8. OLE calls *IDataSource::QueryContinueDrag* on the source side to find out whether the drag operation is still in progress. The MFC *COleDataSource* class responds appropriately.
9. User releases the mouse button to drop the object in the target view window.
10. OLE calls *IDropTarget::OnDrop*, which calls *OnDrop* for the target's view. Because *OnDrop* is passed a *COleDataObject* pointer, it can retrieve the desired format from that object.
11. When *OnDrop* returns in the target program, *DoDragDrop* can return in the source program.

**Chapter-27:**

# Structured Storage

Like Automation and Uniform Data Transfer, structured storage is one of those COM features that you can use effectively by itself. Of course, it's also behind much of the ActiveX technology, particularly compound documents. The *IStorage* interface is used to create and manage structured storage objects. *IStream* is used to manipulate the data contained by the storage object. The *IStorage* and *IStream* interfaces, like all COM interfaces, are simply virtual function declarations. Compound files, on the other hand, are implemented by code in the Microsoft Windows OLE32 DLL. Compound files represent a Microsoft file I/O standard that you can think of as "a file system inside a file."

**Q: Write short note on Structured storage and Compound files?**

**Ans:**

# Storages and the *IStorage* Interface

If you have a storage object, you can manipulate it through the *IStorage* interface. Pay attention to these functions because Microsoft Foundation Class offers no support for storage access. Following are some of the important member functions and their significant parameters.

### HRESULT Commit(…);

Commits all the changes to this storage and to all elements below it.

### HRESULT CopyTo(…, IStorage\*\**pStgDest*);

Copies a storage, with its name and all its substorages and streams (recursively), to another existing storage. Elements are merged into the target storage, replacing elements with matching names.

### HRESULT CreateStorage(const WCHAR\**pName*, …, DWORD *mode*, …, IStorage\*\* *ppStg*);

Creates a new substorage under this storage object.

### HRESULT CreateStream(const WCHAR\**pName*, …, DWORD *mode*, …, IStream\*\* *ppStream*);

Creates a new stream under this storage object.

### HRESULT DestroyElement(const WCHAR\* *pName*);

Destroys the named storage or stream that is under this storage object. A storage cannot destroy itself.

### HRESULT EnumElements(…, IEnumSTATSTG\*\* *ppEnumStatstg*);

Iterates through all the storages and streams under this storage object. The *IEnumSTATSTG* interface has *Next*, *Skip*, and *Clone* member functions, as do other COM enumerator interfaces.

### HRESULT MoveElementTo(const WCHAR\* *pName*, IStorage\* *pStgDest*, const LPWSTR\* *pNewName*, DWORD *flags*);

Moves an element from this storage object to another storage object.

### HRESULT OpenStream(const WCHAR\**pName*, …, DWORD *mode*, …, IStorage\*\* *ppStg*);

Opens an existing stream object, designated by name, under this storage object.

### HRESULT OpenStorage(const WCHAR\**pName*, …, DWORD *mode*, …, IStorage\*\* *ppStg*);

Opens an existing substorage object, designated by name, under this storage object.

### DWORD Release(void);

Decrements the reference count. If the storage is a root storage representing a disk file, *Release* closes the file when the reference count goes to 0.

### HRESULT RenameElement(const WCHAR\* *pOldName*, const WCHAR\* *pNewName*);

Assigns a new name to an existing storage or stream under this storage object.

### HRESULT Revert(void);

Abandons a transaction, leaving the compound file unchanged.

### HRESULT SetClass(CLSID&*clsid*);

Inserts a 128-bit class identifier into this storage object. This ID can then be retrieved with the *Stat* function.

### HRESULT Stat(STATSTG\* *pStatstg*, DWORD *flag*);

Fills in a *STATSTG* structure with useful information about the storage object, including its name and class ID*.*

# Streams and the *IStream* Interface

If you have a stream object, you can manipulate it through the *IStream* interface. Streams are always located under a root storage or a substorage object. Streams grow automatically (in 512-byte increments) as you write to them. An MFC class for streams, *COleStreamFile*, makes a stream look like a *CFile* object. That class won't be of much use to us in this chapter, however.

Once you have a pointer to *IStream*, a number of functions are available to you for manipulating the stream. Here is a list of all the *IStream* functions:

### HRESULT CopyTo(IStream\*\* *pStm*, ULARGE\_INTEGER *cb*, …);

Copies *cb* bytes from this stream to the named stream. *ULARGE\_INTEGER* is a structure with two 32-bit members—*HighPart* and *LowPart*.

### HRESULT Clone(IStream\*\* *ppStm*);

Creates a new stream object with its own seek pointer that references the bytes in this stream. The bytes are not copied, so changes in one stream are visible in the other.

### HRESULT Commit(…);

Transactions are not currently implemented for streams.

### HRESULT Read(void const\* *pv*, ULONG *cb*, ULONG\* *pcbRead*);

Tries to read *cb* bytes from this stream into the buffer pointed to by *pv*. The variable *pcbRead* indicates how many bytes were actually read.

### DWORD Release(void);

Closes this stream.

### HRESULT Revert(void);

Has no effect for streams.

### HRESULT Seek(LARGE\_INTEGER *dlibMove*, DWORD *dwOrigin*, ULARGE\_INTEGER\* *NewPosition*);

Seeks to the specified position in this stream. The *dwOrigin* parameter specifies the origin of the offset defined in the *NewPosition* parameter.

### HRESULT SetSize(ULARGE\_INTEGER *libNewSize*);

Extends or truncates a stream. Streams grow automatically as they are written, but calling *SetSize* can optimize performance.

### HRESULT Stat(STATSTG\* *pStatstg*, DWORD *flag*);

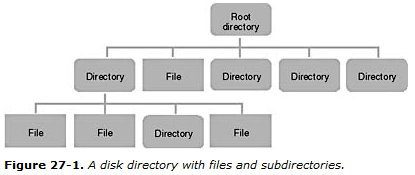
Fills in the *STATSTG* structure with useful information about the stream, including the stream name and size. The size is useful if you need to allocate memory for a read.

### HRESULT Write(void const\* *pv*, ULONG *cb*, ULONG\* *pcbWritten*);

Tries to write *cb* bytes to this stream from the buffer pointed to by *pv*. The variable *pcbWritten* indicates how many bytes were actually written.

# Compound Files

Think of a compound file as a whole file system within a file. Figure 27-1 shows a traditional disk directory as supported by early MS-DOS systems and by Microsoft Windows. This directory is composed of files and subdirectories, with a root directory at the top. Now imagine the same structure inside a single disk file. The files are called streams, and the directories are called storages. Each is identified by a name of up to 32 wide characters in length. A stream is a logically sequential array of bytes, and a storage is a collection of streams and substorages.

****

In a disk file, the bytes aren't necessarily stored in contiguous clusters. Similarly, the bytes in a stream aren't necessarily contiguous in their compound file.

The classic example is a large document composed of chapters and paragraphs within chapters. The document is so large that you don't want to read the whole thing into memory when your program starts, and you want to be able to insert and delete portions of the document. You could design a compound file with a root storage that contains substorages for chapters. The chapter substorages would contain streams for the paragraphs. Other streams could be for index information.

One useful feature of compound files is transactioning. When you start a transaction for a compound file, all changes are written to a temporary file. The changes are made to your file only when you commit the transaction.

# Compound File Fragmentation

Structured storage has a dark side. Like the disk drive itself, compound files can become fragmented with frequent use. If a disk drive becomes fragmented, however, you still have the same amount of free space. With a compound file, space from deleted elements isn't always recovered. This means that compound files can keep growing even if you delete data.

Fortunately, there is a way to recover unused space in a compound file. You simply create a new file and copy the contents. The *IStorage::CopyTo* function can do the whole job in one call if you use it to copy the root storage. You can either write a stand-alone utility or build a file regeneration capability into your application.

**Chapter-27:**

**OLE Embedded Components and Containers**

**Q: What is OLE Embedding container application?**

**Ans:**

# An OLE Embedding Container Application

Now that we've got a working mini-server that supports embedding (EX28A), we'll write a container program to run it. We're not going to use the MFC container support, however, because you need to see what's happening at the OLE interface level. We will use the MFC document-view architecture and the MFC interface maps, and we'll also use the MFC data object classes.

## MFC Support for OLE Containers

If you did use AppWizard to build an MFC OLE container application, you'd get a class derived from *COleDocument* and a class derived from *COleClientItem*. These MFC base classes implement a number of important OLE container interfaces for embedding and in-place activation. The idea is that you have one *COleClientItem* object for each embedded object in a single container document. Each *COleClientItem* object defines a site, which is where the component object lives in the window.

The *COleDocument* class maintains a list of client items, but it's up to you to specify how to select an item and how to synchronize the metafile's position with the in-place frame position. AppWizard generates a basic container application with no support for linking, clipboard processing, or drag and drop. If you want those features, you might be better off looking at the MFC DRAWCLI and OCLIENT samples.

We will use one MFC OLE class in the container—*COleInsertDialog*. This class wraps the *OleUIInsertObject* function, which invokes the standard Insert Object dialog box. This Insert Object dialog enables the user to select from a list of registered component programs.

## Some Container Limitations

Because our container application is designed for learning, we'll make some simplifications to reduce the bulk of the code. First of all, this container won't support in-place activation—it allows the user to edit embedded objects only in a separate window. Also, the container supports only one embedded item per document, and that means there's no linking support. The container uses a structured storage file to hold the document's embedded item, but it handles the storage directly, bypassing the framework's serialization system. Clipboard support is provided; drag-and-drop support is not. Outside these limitations, however, it's a pretty good container!

## Container Features

So, what does the container actually do? Here's a list of features:

* As an MFC MDI application, it handles multiple documents.
* Displays the component's metafile in a sizeable, moveable tracker rectangle in the view window.
* Maintains a temporary storage for each embedded object.
* Implements the Insert Object menu option, which allows the user to select a registered component. The selected component program starts in its own window.
* Allows embedded objects to be copied (and cut) to the clipboard and pasted. These objects can be transferred to and from other containers such as Microsoft Word and Microsoft Excel.
* Allows an embedded object to be deleted.
* Tracks the component program's loaded-running transitions and hatches the tracker rectangle when the component is running or active.
* Redraws the embedded object's metafile on receipt of component change notifications.
* Saves the object in its temporary storage when the component updates the object or exits.
* Copies the embedded object's temporary storage to and from named storage files in response to Copy To and Paste From commands on the Edit menu.

**Chapter-29:**

**Introducing the Active Template Library**

**Q: Write short note on ATL?Or**

**Explain ATL in detail?**

**Ans:**

Microsoft Visual C++—the Active Template Library (ATL). You'll start by quickly revisiting the Component Object Model (COM). The Active Template Library, focusing first on C++ templates and raw C++ smart pointers and how they might be useful in COM development.

## 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.

## ActiveX, MFC, and COM

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.

**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.

## 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.

## 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. 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.

**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. 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.

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.

**Q: Write steps to create server-side ATL programming?**

**Ans:**

## ATL COM AppWizard Options

In the Step 1 dialog, 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.

Selecting DLL as the server type produces all the necessary pieces to make your server DLL fit into the COM milieu. 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.

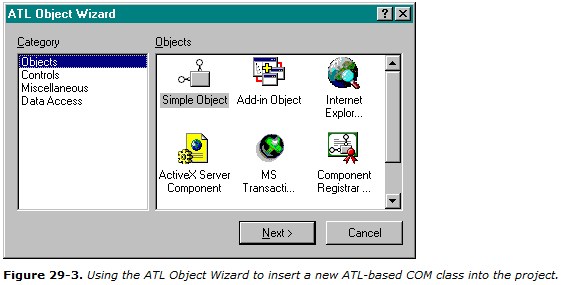
These products include:

* 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 29-3. 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).



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

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.

## 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*.

**Q: Explain the Basic ATL Architecture?**

**Ans:**

## Basic ATL Architecture

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, 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.

## 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. *CComObjectRootBase* 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.

## 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*'sdefinition—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.

**Chapter-30:**

**ATL and ActiveX Controls**

**Q: Write steps to create a control using ATL?**

**Ans:**

ActiveX controls are small gadgets (usually UI-oriented) written around the Component Object Model. There are several steps involved in creating an ActiveX control using ATL, including:

* Deciding what to draw
* Developing incoming interfaces for the control
* Developing outgoing interfaces (events) for the control
* Implementing a persistence mechanism for the control
* Providing a user interface for manipulating the control's properties

**ActiveX Controls**

Even today, there's some confusion as to what really constitutes an ActiveX control. In 1994, Microsoft tacked some new interfaces onto its Object Linking and Embedding protocol, packaged them within DLLs, and called them OLE Controls. Originally, OLE Controls implemented nearly the entire OLE Document embedding protocol. In addition, OLE Controls supported the following:

* Dynamic invocation (Automation)
* Property pages (so the user could modify the control's properties)
* Outbound callback interfaces (event sets)
* Connections (a standard way to for clients and controls to hook up the event callbacks)

When the Internet became a predominant factor in Microsoft's marketing plans, Microsoft announced its intention to plant ActiveX Controls on Web pages. At that point, the size of these components became an issue. Microsoft took its OLE Control specification, changed the name from OLE Controls to ActiveX Controls, and stated that all the features listed above were optional. This means that under the new ActiveX Control definition, a control's only requirement is that it be based on COM and that it implement *IUnknown*. Of course, for a control to be useful it really needs to implement most of the features listed above. So in the end, ActiveX Controls and OLE Controls refer to more or less the same animal.

# Using ATL to Write a Control

Although creating an ActiveX control using ATL is actually a pretty straightforward process, using ATL ends up being a bit more burdensome than using MFC. That's because ATL doesn't include all of MFC's amenities. For example, ATL doesn't include device context wrappers. When you draw on a device context, you need to use the raw device context handle. In addition, ClassWizard doesn't understand ATL-based source code, so when you want your control to handle messages, you end up using the "TypingWizard".

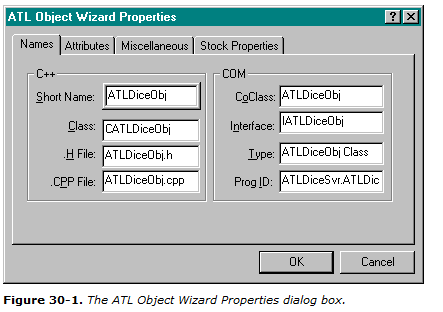
Despite these issues, creating an ActiveX control using ATL is a whole lot easier than creating one from scratch. Also, using ATL gives you a certain amount of flexibility you don't get when you use MFC. For example, while adding dual interfaces to your control is a tedious process with MFC, you get them for free when you use ATL. The ATL COM Object Wizard also makes adding more COM classes (even noncontrol classes) to your project very easy, while adding new controls to an MFC-based DLL is a bit more difficult.

## Creating the Control

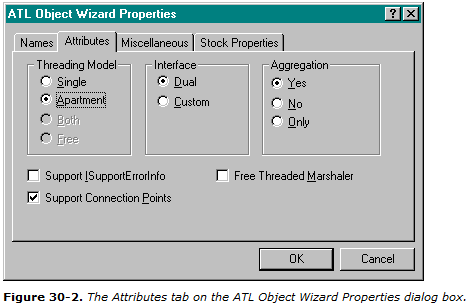
As always, the easiest way to create a COM server in ATL is to use the ATL COM Object Wizard. To use the ATL COM Object Wizard, select New from the File menu. Select the Project tab in the New dialog, and highlight the ATL COM AppWizard item. Name the project something clever like *ATLDiceSvr*. As you step through AppWizard, just leave the defaults checked. Doing so will ensure that the server you create is a DLL.

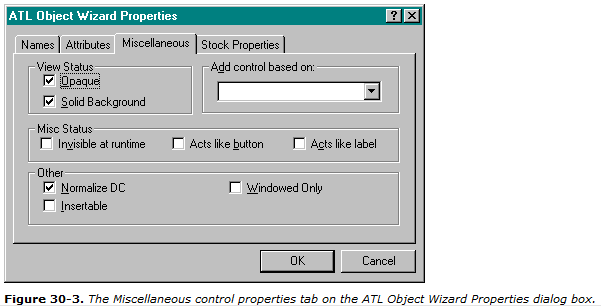
Once the DLL server has been created, perform the following steps:

1. Select New ATL Object from the Insert menu to insert a new ATL object into the project.
2. In the ATL Object Wizard, select Controls from the Category list and then select Full Control from the Objects list.
3. Click Next to open the ATL Object Wizard Properties dialog. In the Short Name text box on the Names tab, give the control some clever name (like *ATLDiceOb*). The dialog box should look similar to Figure 30-1.

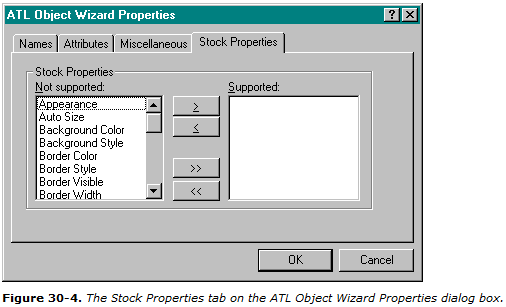
****

1. Select the Attributes tab. Here's where you configure the control. For example, you can
   * Designate the threading model for the control
   * Decide whether the main interface is a dual or custom interface
   * Indicate whether your control supports aggregation
   * Choose whether you want to use COM exceptions and connection points in your control
2. To make your life easier for now, select Support Connection Points. (This will save you some typing later on.) Leave everything else as the default value. Figure 30-2 shows what the Attributes tab on the ATL Object Wizard Properties dialog box looks like now.
3. Select the Miscellaneous tab. Here you have the option of applying some miscellaneous traits to your control. For example, you can give the control behaviors based on regular Microsoft Windows controls such as buttons and edit controls. You might also select other options for your control, such as having your control appear invisible at runtime or giving your control an opaque background. Figure 30-3 shows the available options.

****

****

1. Finally, select the Stock Properties tab if you want to give your control some stock properties. Stock properties are those properties that you might expect any control to have, including background colors, border colors, foreground colors, and a caption. Figure 30-4 shows the Stock Properties tab.

****

1. When you've finished selecting the attributes for the control, click OK.

**Q: Explain ATL’s Control Architecture?Or**

**Write short note on ATL’s Control Architecture?**

**Ans:**

## ATL's Control Architecture

From the highest level, an ActiveX control has two aspects to it: its external state (what it renders on the screen) and its internal state (its properties). Once an ActiveX control is hosted by some sort of container (such as a Microsoft Visual Basic form or an MFC dialog box), it maintains a symbiotic relationship with that container. The client code talks to the control through incoming COM interfaces such as *IDispatch* and OLE Document interfaces like *IOleObject* and *IDataObject*.

The control also has the opportunity to talk back to the client. One method of implementing this two-way communication is for the client to implement an *IDispatch* interface to represent the control's event set. The container maintains a set of properties called ambient properties that the control can use to find out about its host. For instance, a control can camouflage itself within the container because the container makes the information stored in these properties available through a specifically named *IDispatch* interface. The container can implement an interface named *IPropertyNotifySink* to find out when the properties within a control might change. Finally, the container implements *IOleClientSite* and *IOleControlSite* as part of the control-embedding protocol.

### *CComControl*

You can find the definition of *CComControl* in Microsoft's ATLCTL.H file under ATL's Include directory. *CComControl* is a template class that takes a single class parameter:

template <class T>

class ATL\_NO\_VTABLE CComControl : public CComControlBase,

public CWindowImpl<T>

{

.

.

.

};

*CComControl* is a rather lightweight class that does little by itself—it derives functionality from *CComControlBase* and *CWindowImpl*. *CComControl* expects the template parameter to be an ATL-based COM object derived from *CComObjectRootEx*. *CComControl* requires the template parameter for various reasons, the primary reason being that from time to time the control class uses the template parameter to call back to the control's *InternalQueryInterface*.

*CComControl* implements several functions that make it easy for the control to call back to the client. For example, *CComControl* implements a function named *FireOnRequestEdit* to give controls the ability to tell the client that a specified property is about to change. This function calls back to the client through the client-implemented interface *IPropertyNotifySink*. *FireOnRequestEdit* notifies all connected *IPropertyNotifySink* interfaces that the property specified by a certain *DISPID* is about to change.

*CComControl* also implements the *FireOnChanged* function. *FireOnChanged* is very much like *FireOnRequestEdit* in that it calls back to the client through the *IPropertyNotifySink* interface. This function tells the clients of the control (all clients connected to the control through *IPropertyNotifySink*) that a property specified by a certain *DISPID* has already changed.

In addition to mapping the *IPropertyNotifySink* interface to some more easily understood functions, *CComControl* implements a function named *ControlQueryInterface,* which simply forwards on to the control's *IUnknown* interface. (This is how you can get a control's *IUnknown* interface from inside the control.) Finally, *CComControl* implements a function named *CreateControlWindow*. The default behavior for this function is to call *CWindowImpl::Create*.

### *CComControlBase*

*CComControlBase* is a much more substantial class than *CComControl*. To begin with, *CComControlBase* maintains all the pointers used by the control to talk back to the client. *CComControlBase* uses ATL's *CComPtr* smart pointer to include member variables that wrap the following interfaces implemented for calling back to the client:

* + A wrapper for *IOleInPlaceSite*(*m\_spInPlaceSite*)
  + An advise holder for the client's data advise sink (*m\_spDataAdviseHolder*)
  + An OLE advise holder for the client's OLE advise sink (*m\_spOleAdviseHolder*)
  + A wrapper for *IOleClientSite* (*m\_spClientSite*)
  + A wrapper for *IAdviseSink* (*m\_spAdviseSink*)

*CComControlBase* also uses ATL's *CComDispatchDriver* to wrap the client's dispatch interface for exposing its ambient properties.

*CComControlBase* is also where you'll find the member variables that contain the control's sizing and positioning information: *m\_sizeNatural*, *m\_sizeExtent*, and *m\_rcPos*. The other important data member within *CComControlBase* is the control's window handle. Most ActiveX controls are UI gadgets and as such maintain a window. *CWindowImpl* and *CWindowImplBase* handle the windowing aspects of an ATL-based ActiveX control.

### *CWindowImpl* and *CWindowImplBase*

*CWindowImpl* derives from *CWindowImplBase*, which in turn derives from *CWindow* and *CMessageMap*. As a template class, *CWindowImpl* takes a single parameter upon instantiation. The template parameter is the control being created. *CWindowImpl* needs the control type because *CWindowImpl* calls back to the control during window creation. Let's take a closer look at how ATL handles windowing.

### ATL Windowing

Just as *CComControl* is relatively lightweight (most work happens in *CComControlBase*), *CWindowImpl* is also relatively lightweight. *CWindowImpl* more or less handles only window creation. In fact, that's the only function explicitly defined by *CWindowImpl*. *CWindowImpl::Create* creates a new window based on the window class information managed by a class named *\_ATLWNDCLASSINFO*.

### ATL Message Maps

The root of ATL's message mapping machinery lies within the *CMessageMap* class. ATL-based controls expose message maps by virtue of indirectly deriving from *CWindowImplBase*. In MFC, by contrast, deriving from *CCmdTarget* enables message mapping. However, just as in MFC, it's not enough to derive from a class that supports message maps. The message maps actually have to be there—and those message maps are implemented via macros.

To implement a message map in an ATL-based control, use message map macros. First ATL's *BEGIN\_MSG\_MAP* macro goes into the control class's header file. *BEGIN\_MSG\_MAP* marks the beginning of the default message map. *CWindowImpl::WindowProc* uses this default message map to process messages sent to the window. The message map directs messages either to the appropriate handler function or to another message map. ATL defines another macro named *END\_MSG\_MAP* to mark the end of a message map. Between *BEGIN\_MSG\_MAP* and *END\_MSG\_MAP* lie some other macros for mapping window messages to member functions in the control. For example, here's a typical message map you might find in an ATL-based control:

BEGIN\_MSG\_MAP(CAFullControl)

CHAIN\_MSG\_MAP(CComControl<CAFullControl>)

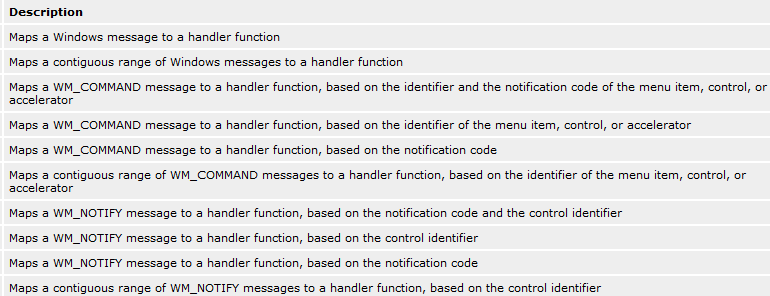
DEFAULT\_REFLECTION\_HANDLER()

MESSAGE\_HANDLER(WM\_TIMER, OnTimer);

MESSAGE\_HANDLER(WM\_LBUTTONDOWN, OnLButton);

END\_MSG\_MAP()

This message map delegates most of the message processing to the control through the *CHAIN\_MSG\_MAP* macro and handles message reflection through the *DEFAULT\_REFLECTION\_HANDLER* macro. The message map also handles two window messages explicitly: WM\_TIMER and WM\_LBUTTONDOWN. These are standard window messages that are mapped using the *MESSAGE\_HANDLER* macro. The macros simply produce a table relating window messages to member functions in the class. In addition to regular messages, message maps are capable of handling other sorts of events. Here's a rundown of the kinds of macros that can go in a message map.



Handling messages within ATL works much the same as in MFC. ATL includes a single window procedure through which messages are routed. Technically, you can build your controls effectively without understanding everything about ATL's control architecture. However, this knowledge is sometimes helpful as you develop a control, and it's even more useful when debugging a control.