INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

A Survey on Microsoft Component-based Programming Technologies

Junping Li

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

November 1999

© Junping Li, 1999

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-47848-3

Canada

Abstract

A Survey on Microsoft Component-based Programming Technologies

Junping Li

Component-based programming is the trend in the software industry. Software components are reusable software units in executable form that can be plugged into other components from other vendors with relatively little effort. The components present a set of functionality through interfaces.

This report will discuss the Microsoft component-based programming technologies such as DLL, COM/DCOM, OLE Automation and ActiveX Controls. The motivation for establishing each of these techniques, the fundamental architecture, and the implementation examples for creating software objects using these techniques will be discussed.

Acknowledgement

I would like to thank Dr. Lixin Tao for being my project advisor. His guidance is greatly appreciated.

My thanks also go to my wife, Jingyi, and my sons Simon and Raymond, for their patience.

TABLE OF CONTENTS

1 INTRODUCTION **1**

2 DYNAMIC LINK LIBRARIES **3**

2.1 THE MOTIVATION OF USING DLLS 3
2.2 THE ADVANTAGES OF USING DLLS 3
2.3 HOW TO BUILD A DLL 4
2.4 HOW TO LINK A DLL 7
2.4.1 Link to a DLL Statically *8*
2.4.2 Link to a DLL Dynamically *9*
2.4.3 Link to a DLL in a Visual Basic Client *10*

3 COM/DCOM **11**

3.1 WHAT IS COM? 11
3.2 WHY IS COM? 11
3.3 THE REQUIREMENTS FOR A COMPONENT 12
3.4 WHAT IS AN INTERFACE? 12
3.5 QUERYING FOR INTERFACE 14
3.6 THE CONTROL OF COMPONENTS LIFETIME 16
3.7 DYNAMIC LINKING OF COMPONENTS 19
3.8 SOME COM DCOM TYPES AND THE REGISTRY 20
3.8.1 HRESULT *20*
3.8.2 GUID *21*
3.8.3 The Windows Registry *22*
3.9 IN-PROC SERVER 24
3.9.1 Class Factory *25*
3.9.2 In-proc Server DLL Entry Points *28*
3.9.3 Register/Unregister Servers *29*
3.9.4 Example of an In-proc Server Component *29*
3.10 CONTAINMENT AND AGGREGATION 30
3.11 LOCAL SERVER 32
3.11.1 RPCs *33*
3.11.2 Marshaling *33*
3.11.3 Proxy Stub DLLs *34*
3.11.4 IDL *35*
3.12 REMOTE SERVER 40
3.12.1 Use Local Server Remotely *40*
3.12.2 Use Remote Server Explicitly *42*

4 OLE AUTOMATION **44**

4.1 WHAT IS OLE? 44
4.2 AUTOMATION 46
4.3 IDISPATCH 47
4.4 DISPINTERFACE 50
4.5 DUAL INTERFACE 50
4.6 USING IDISPATCH INTERFACE 50
4.7 METHODS AND PROPERTIES 52
4.8 VARIANTS 54
4.9 BSTRS 56
4.10 SAFEARRAYS 57
4.11 TYPE LIBRARIES 58
4.12 IMPLEMENTING IDISPATCH 62
4.13 EXCEPTION HANDLING 64

5 ACTIVEX CONTROLS **66**
5.1 ACTIVEX CONTROL OVERVIEW 66
5.1.1 What is an ActiveX Control? *66*
5.1.2 Characteristics of ActiveX Controls *66*
5.1.3 Features of ActiveX Controls *68*
5.2 TOOLS FOR CREATING ACTIVEX CONTROLS 78
5.2.1 Microsoft Tools *78*
5.2.2 Creating a Skeleton Project *79*
5.2.3 The ControlWizard Created Files *81*
5.2.4 The Control Module Class *82*
5.2.5 The Control Class *83*
5.2.6 The Property Page Class *87*
5.2.7 Test the Control *88*
5.3 PROPERTIES 89
5.3.1 Ambient Properties *89*
5.3.2 Stock Properties *90*
5.3.3 Adding Stock Properties *91*
5.3.4 Adding Custom Properties *92*
5.4 PROPERTY PERSISTENCE 93
5.5 METHODS 94
5.6 EVENTS 96
5.7 PROPERTY PAGES 100

6 CONCLUSIONS **103**

REFERENCES: **104**

APPENDIX 1 A DLL EXAMPLE **105**

APPENDIX 2 AN CLIENT EXAMPLE USING A DLL STATICALLY **107**

APPENDIX 3 A CLIENT EXAMPLE USING A DLL DYNAMICALLY **108**

APPENDIX 4 A VISUAL BASIC EXAMPLE USING A DLL **109**

APPENDIX 5 AN INTERFACE EXAMPLE **110**

APPENDIX 6 AN EXAMPLE TO QUERY INTERFACES **111**

APPENDIX 7 A COMPONENT LIFETIME CONTROL EXAMPLE **115**

APPENDIX 8 USE A COMPONENT IMPLEMENTED IN A DLL **118**

APPENDIX 9 AN IN-PROC SERVER COMPONENT EXAMPLE **124**

APPENDIX 10 AN IN-PROC SERVER COMPONENT CLIENT EXAMPLE **135**

APPENDIX 11 AN ACTIVEX CONTROL EXAMPLE **137**

1 Introduction

Today's software applications become more and more complex and they require more and more time and resources to develop. They are difficult and costly to maintain. In addition, most of the software applications are still monolithic. They are developed with many features, each of them is usually related to many of others. Each of these features cannot be easily removed, upgraded or replaced with alternatives independently. Further, applications are not easily integrated with each other. The data and functionality of one application is not readily available to other applications.

Software reusability has long been discussed to solve the problems faced by the software industry. Currently, the Object-oriented Programming (OOP) technique is used widely to achieve software reusability. C++ is one of the mostly used OOP languages nowadays.

While Object-oriented Programming is powerful, it still suffers from some major problems. Software objects created by different vendors cannot interact with one another through a standard framework in the same address space, in different address spaces or on different machines. OOP software reuse is still at the source code level and subject to breaking if any of the base-classes in an inheritance hierarchy changes its interface.

The solution to these problems is to create reusable software components. Software components are reusable software units in executable form that can be plugged into other components from other vendors with relatively little effort. The components present a set of functionality through interfaces.

The component-based programming is the trend in the software industry. Various techniques have been developed. Among them, Microsoft has made a lot of effort to establish a series of component-based architectures. The techniques on Dynamic Link Library (DLL), Component Object Model (COM)/Distributed Component Object Model (DCOM), Object Linking and Embedding (OLE) and ActiveX all contribute to the component-based programming.

This survey will go through the basics of Microsoft component-based programming techniques such as DLL, COM/DCOM, OLE Automation and ActiveX Control. It discusses the motivation for establishing each of these techniques, the fundamental architecture, and the implementation examples for creating software objects using these techniques.

2 Dynamic Link Libraries

DLL technique is the foundation of Microsoft component-based technologies.

2.1 The Motivation of Using DLLs

Very often, a function in a library will be used in many applications. If the function is statically linked to them when the applications running concurrently, then the code is replicated and memory is allocated multiple times for the same function. This problem will happen not only on functions used by different applications running concurrently, but also applies to any functions used by a single application that has multiple instances loaded at the same time. This results in memory waste and makes the size of application executables bigger. Furthermore, it makes the application monolithic as every function it uses is built in a single executable file. This leads us to ask: can we allow multiple applications that are running at the same time to share the same library functions? The answer is the Dynamic Link Libraries (DLLs).

2.2 The Advantages of Using DLLs

A Dynamic Link Library (DLL) is an executable file to be used as a library. When multiple applications are using the same DLL, only one copy of the DLL is maintained in the memory. The linker maps the DLL into the address space of the executable that uses it. Using DLLs will result in the following advantages[1]:

- Using DLLs reduces the memory used when multiple processes are accessing the DLL simultaneously because a single copy of the DLL resides in the memory. This

results in less virtual memory swapping and enhances the performance of the applications that use the DLL.

- As long as the function prototype, i.e., its interface, is not changed, changing the function's implementation in a DLL will not require to recompile or relink the applications that use the DLL's function. However, if the function is statically linked to an application and the function is changed, the application has to be relinked. This decouples the software clients from their servers' implementation and makes the applications more component-oriented.
- Since a DLL's implementation can be modified without affecting the applications that use it, the vendor of the DLL can make additional support after the DLL is shipped. For instance, a display driver DLL can be updated incrementally to support new types of the display.
- Applications written in different programming languages can use the same function in a DLL as long as the applications follow the same calling convention as specified for the function in the DLL. This will break the programming language boundary and make it possible that a component can be used by different programs regardless what languages they are written.

2.3 How to Build a DLL

Certainly, the first step in using a DLL is to build a DLL. It is easier to build a DLL by using a programming tool. For instance, in Visual C++, version 6.0, going through the following steps will create a new DLL project:

1. Select New from the File menu and click the Projects tab, choosing "Win32 Dynamic-Link Library";
2. In Project name edit box, type the DLL name, e.g., Example, then specify the directory where the DLL project will be created;
3. In the step 1 Wizard dialog, select "An empty DLL project", then finish the Wizard by clicking the Finish button; The Wizard will then create the files used in the DLL project.

Once the DLL project is created, the DLL source files must be added to the project. This can be done by selecting Add to Project from the Project menu, then selecting Files to invoke the dialog to add the files.

A DLL can contain variables and functions to be used by its clients. A DLL can also contain variables and functions used by the DLL internally. The variables and functions that can be used outside the DLL must be exported.

There are two ways to mark the symbols in a DLL as exported:

1. Use a definition file in the DLL project;
2. Use storage class modifiers to declare the symbols;

A DLL's definition file has the same name as the DLL, but it has the file extension DEF instead of DLL. For example, a DLL called Example.dll can have a DLL definition file called Example.def. A variable or a function can be marked as exported in the EXPORTS

section of the DLL's definition file. The following code is an example of a DLL's definition file:

Example.DEF

; Example.dll's module definition file

LIBRARY Example.dll
EXPORTS
ExampleFunction @1

In this example, a DLL called Example.dll implements a function called ExampleFunction. This function is exported by specifying it in the DLL's definition file called Example.def.

The other way to export a variable or a function in a DLL, which is available in 32-bit Windows, is to use __declspce(dllexport) extended storage class modifier defined in the Microsoft C++ language. You can mark a function or data as being exported to applications and other DLLs with this modifier, e.g.,

```
__declspec(dllexport) int ExportedFunction(int arg);
__declspec(dllexport) int j;
```

C++ compiler will decorate or mangle the function names to allow the overloading of methods. If a function is exported as a C++ function, the decorated name will be exported. This will not be a problem if the client of the DLL and the DLL itself will use the same C++ compiler. To remove this limitation, we can turn off the name decoration by marking the function with extern "C" or use

extern "C" { }

to turn off the name decoration for a block of code.

The client of the DLL that will uses the DLL's exported variables or functions will have to mark the variables or functions with __declspec(dllimport) modifier. e.g..

__declspec(dllimport) int ExportedFunction(int arg);
__declspec(dllimport) int j;

The DLL author should provide the header file. which contains the imported variables or functions marked with __declspec(dllimport) modifier. of the DLL to be used by the DLL's clients. The same header file. which contains the exported variables or functions marked with __declspec(dllexport) modifier. can be used by the DLL implementation file as well. Appendix 1 gives the list of the header and the implementation files of a simple DLL using the __declspec modifier.

An exported function in a DLL can also take or return parameters like a function in an EXE program. For example. the second exported function in the Appendix 1 example DLL. dllParamFunction. takes an integer. a string. an array of 3 float numbers. passes back the sum of the 3 array elements to the caller and returns 1 to indicate success.

2.4 How to link a DLL

When a DLL is ready to use, the clients of the DLL can use its exported data or functions. If the DLL is linked statically. the header file containing the imported data and functions must be included in the client program. Furthermore. the program must be able to access the DLL library. The easiest way of doing this is to place the DLL in the same directory where the client program is located. Alternatively. a path to the DLL can be specified in

the Windows environment. A DLL can be linked either statically or dynamically by a client application.

Following the steps below will create a console application in Visual C++ version 6.0 that can be used as a client to use the DLL:

1. Select New from the File menu, click the Projects tab, choosing "Win32 Console Application";
2. Specify the name for the application in the Project name edit box and the directory where the application will be created in the Location edit box;
3. In the Wizard dialog, choose "An empty project", then clicking the Finish button;

2.4.1 Link to a DLL Statically

To statically link to a DLL, also known as implicit linking, the linker links to an import library of the DLL that was generated by the compiler that created the DLL. For instance, when Example.dll was built, a companion library file called Example.lib was generated as well. The client of the Example.dll links statically to Example.lib. This .lib file essentially consists of the stub functions that forward the calls on to the DLL. In this way, the client uses the DLL as if it uses the static library. In implicit linking, the DLL is loaded when the client EXE file is started and it is released when the client EXE exits. Certainly, the client program that uses the DLL's exported functions must include the header file containing the declarations of the functions and the client application must specify Example.lib as one of its linked libraries.

The console application example listed in Appendix 2 shows a client written in C, which links to the Appendix 1 example DLL statically. When you run this client, the example DLL is loaded into the client's address space. When running the client program, it calls the first DLL's exported function which simply pops up a message box saying that "The example DLL API is called!" as specified in the implementation of function, dllExampleFunction, in dllExample.c. The client then passes an integer 12345, a string "Example String Parameter", an array of 3 real values, 1.1, 2.2, 3.3, to the second DLL function, dllParamFunction. The function in the DLL will print out the passed-in values, calculate the sum and pass it back to the client. The client then checks the returned value from the DLL function and print out the calculated sum value on success.

2.4.2 Link to a DLL Dynamically

A DLL can also be loaded when it is needed. This kind of linking is known as dynamic linking, or explicit linking. To dynamically link to a DLL, there is no need to link to the companion .lib file of the DLL when the compiler generates the client application. Instead, when the client needs to call a function in a DLL, it will call the Windows API:

LoadLibrary("dllFileName")

to explicitly load the DLL and then, use the Windows API

GetProcAddress(dllModuleHandle, "functionName")

to get the function pointer. Use the function pointer to call the function. When the DLL is no longer needed, call the Windows API

FreeLibrary(dllModuleHandle)

to release the DLL explicitly.

The console example listed in Appendix 3 shows a client application written in C. which loads the Appendix 1 example DLL when it wants to use the DLL's exported function. Upon success. it gets the exported function address and calls the function and finally. it releases the DLL.

2.4.3 Link to a DLL in a Visual Basic Client

The services of a DLL can also be used by the clients written in other programming languages. However. the calling convention of the client must conform to the calling convention of the exported functions in the DLL that the client uses. Generally. the functions exported from a DLL are C functions in standard calling convention. also known as Pascal calling convention. where the called functions clean up the stack frames.

The example listed in Appendix 4 shows a client application written in Visual Basic which uses the Appendix 1 example DLL's exported function. It uses the Visual Basic Declare statement to define the function name and the DLL name where the function resides. When the user clicks the Use DLL... button. the handler of the event will call the example DLL's function. which simply pops up a message box. If the user clicks Exit button. the program exits.

3 COM/DCOM

Component Object Model(COM)/Distributed Component Object Model(DCOM) is the fundamental architecture on Microsoft's component-based programming technologies.

3.1 What is COM?

Components are defined as small executable files that provide certain services. Different components can be glued together to create complex software applications. An application using a component is called a client of the component. Components are linked to their clients dynamically. Therefore, the client applications do not need to be recompiled or relinked if the components they use are changed. COM/DCOM is the technology developed in Microsoft for developing software components.

COM/DCOM itself is a specification[2] and any programming language can be used to write COM/DCOM components. COM/DCOM is the base for other Microsoft technologies such as OLE, ActiveX.

3.2 Why is COM?

Why does Microsoft develop COM/DCOM? Currently, a software application is just a big executable file. The services provided by different applications cannot be shared. Further, to incorporate any change, the application must be recompiled and relinked. To solve these problems, we can break the application into separate, smaller components. The benefits of using components can be listed as follows:

- A new application can be built by assembling existing components. This speeds up the developing of new applications;
- An application can be enhanced by replacing the old components with the new ones;
- An application can be customized by plugging in new components or unplugging the components they do not need;

3.3 The Requirements for a Component

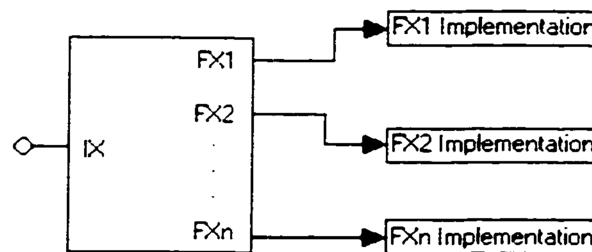
Since a component must have the capability to plug and unplug from an application dynamically, it must satisfy the following requirements[3]:

1. A component must be linked dynamically;
2. Components can be written by any programming language and can be used by applications written in any language;
3. Components must be shipped in executable files and ready to use. That is, they have to be in DLL form if they are used in the same address space of their clients or in EXE form if they are used in different address space of their clients;
4. New versions of a component should work with both old and new clients;
5. A component and their clients should be able to run in the same process, in different processes, or on different machines;

3.4 What is an Interface?

A component provides certain services. These services are specified by interfaces. An interface is a contract between two different software objects. In COM/DCOM, an

interface is defined as a set of related function declarations. A component supports an interface by implementing the functions in the interface. More precisely, at low level, an interface in COM/DCOM is an array of memory addresses, each array element is an address of a function implemented by the component. The following diagram shows an example interface IX.



The Representation of an Example Interface IX

From the point of view of clients, a component is just a set of interfaces. Anything the clients do to the component must be through the interfaces that the component provides and implements.

Any programming language and data structure can be used to define interfaces. Appendix 5 lists an example from Rogerson[3] which defines two interfaces in C++ using two pure abstract base classes. In C++, any class that inherits from a pure abstract base class must implement the pure virtual functions declared in the base class. Hence, the derived class can be used to define a component that supports the interface defined by the pure abstract base class.

3.5 Querying for Interface

A component provides its services through the interfaces that it supports. Therefore, if a client of the component wants to use the services provided by a component, the client will have to get the interface that defines the services. To enable the clients to get the interfaces supported by a component, COM/DCOM specifies that all interfaces must inherit from a special interface called IUnknown. The IUnknown interface is declared as follows in unknwn.h:

```
interface IUnknown
{
   virtual HRESULT __stdcall QueryInterface(const IID& iid,
                                            void** ppv) = 0;
   virtual ULONG __stdcall AddRef() = 0;
   virtual ULONG __stdcall Release() = 0;
};
```

In other words, every interface must have QueryInterface, AddRef and Release as its first three functions. Since a component is just an implementation of a set of interfaces, a component must implement these three functions. In short, any COM/DCOM interface is an IUnknown interface. Any component must implement QueryInterface, AddRef and Release functions.

To get an interface from a component, a client must call the component's QueryInterface function. The first parameter taken by the QueryInterface function is the id of the interface to be queried. It is an IID(interface identifier) structure which uniquely identifies the interface. The second parameter the function takes is the returned address of the requested interface pointer. The QueryInterface function must either return S_OK or

E_NOINTERFACE. The return value must be compared in the clients with a macro SUCCEEDED or FAILED to find out if the requested interface is found or not.

The implementation of the QueryInterface function must obey some rules so that the interface queries can be performed and the clients can know about the components' interfaces in a deterministic way. Rogerson[3] listed the following rules:

1. The same IUnknown interface pointer must always be returned whenever the IUnknown interface for a component is queried;
2. The query of an interface on an instance of a component must always succeed or always fail;
3. An interface queried from itself must get itself;
4. If you can get interface A from interface B, you must be able to get interface B from interface A;
5. If you can get an interface from a component, you must be able to get that interface from any interface that the component supports.

Appendix 6 is an example of QueryInterface implementation[3] and usage. This example shows a component that supports two interfaces IX and IY, both inherit from the required IUnknown interface. The component implements the QueryInterface function in a manner that satisfies the COM/DCOM rules.

You can see that the IUnknown interface pointer returned from the QueryInterface function will always be the same 'this' pointer casted to IX pointer. The same is true in the CreateInstance function. The 'this' pointer cannot be casted to IUnknown pointer as it

will be ambiguous, since both IX and IY inherit from IUnknown. To make IUnknown pointer always the same, it must be casted consistently to IX or to IY. In this example, the implementation of the functions of IUnknown is the same when calling the functions through IX or through IY. However, casting to IY will cause the change of the pointer value by the C++ compiler, which might give problem when cleaning up the memory pointed to by the pointer. Another observed fact is that the queries for IUnknown, IX and IY (or in terms of IIDs, IID_IUnknown, IID_IX and IID_IY) will always succeed and it will always fail for other interfaces such as IZ. It is shown that querying an interface from the interface itself from the example component's QueryInterface function will get the same interface. The example also shows that after interface IY is retrieved from IX interface, IX can also be retrieved from IY. It is also apparent that a supported interface can be retrieved through any supported interface of the component in the example. This example also demonstrates that the client of the component uses SUCCEEDED macro to test if an interface is supported in the component.

Since the supported interfaces define a component and QueryInterface defines what interfaces that a component supports, the implementation of QueryInterface defines the component.

3.6 The Control of Components Lifetime

As for any software object, the lifetime control is important. When can a component be removed from the memory? Only when the component is not being used by any of its

clients. Since a client of the component only knows the components' interfaces it is using, it is very hard for a client to know if the component is being used by other clients.

The strategy is to let the component itself take care of its lifetime. In COM/DCOM, a memory management technique, reference counting, is used. In this technique, the component keeps a reference count number. Each time a client gets an interface from the component, the reference count is incremented. When a client is finished with using an interface, the reference count is decremented. When the reference count becomes zero, which means no interface of the component is being used, the component deletes itself from the memory. This is exactly what the two other member functions of IUnknown interface, AddRef and Release, are for. As the name implies, AddRef is used to increment the reference count and Release is used to decrement the reference count.

When implementing reference counting, certain rules must be followed[3]:

1. Whenever an interface is returned, either as a return value of the function or as an out parameter of the function, AddRef should be called by the function;
2. Whenever an interface is no longer used, Release should be called;
3. Whenever a new reference to an interface is created such as an interface pointer is assigned to another interface pointer, AddRef should be called;
4. An interface pointer passed as an in parameter in a function does not need to call AddRef and Release as the function is nested inside the lifetime of the caller;

5. A function must call Release on an interface pointer passed as an in-out parameter before overwriting the in parameter with another interface pointer. The function must also call AddRef on the out parameter before the function returns;
6. Local variables as the copies of the interface pointers are not required to call the AddRef and Release as local variables only exist for the lifetime of the function;
7. Always use AddRef and Release pairs whenever in doubt.

In short, the AddRef function should be called for every new copy of an interface pointer of a component. Each AddRef function should be paired with a Release function call.

The client should also treat each interface as if it has its own reference count although the component may implement a single reference count for all interfaces it supports.

Appendix 7 is an example component[3] with lifetime control implementation. In this example, the component keeps a reference count variable. The AddRef function is implemented using the Win32 API InterlockedIncrement and the Release function is implemented using InterlockedDecremented API. These APIs will only allow one thread to access the variable at a time. The component will also delete itself in the Release function if the reference count reaches 0. The return values from AddRef and Release function are the new reference count. However, the clients should not depend on these return values as these values may not be stable. These return values should only be used for debugging purpose.

In the example, the AddRef is called on an interface whenever a copy of the interface is made such as in QueryInterface and CreateInstance functions. The client calls Release function on an interface whenever it finishes using the interface. Note that the client is not bothered with deleting the component at all. The component deletes itself when no interface is being used.

3.7 Dynamic Linking of Components

One basic requirement in COM is that changing component should not affect the clients that use it. This requires that the component must be in a ready-to-use executable form. We can use a DLL to contain components. A DLL is in the same address space as the application it is linked to. For this reason, DLLs are also called in-process server, or in-proc server.

How to dynamically link a component to the client? As mentioned in the DLL section, the component has to be built in a DLL project and the functions called directly by the clients must be exported.

Appendix 8 is an example of a component implemented in a DLL and a client using the component. In this example, the component and the client are in two separate executables. The component is in a DLL and the client is in an EXE file. When the client wants to use the component, it loads the DLL that contains the component and then creates the component by calling the component's exported function CreateInstance. This

function returns an IUnknown interface pointer, which will be used by the client to query for other interfaces and use the functions in the interfaces.

3.8 Some COM/DCOM Types and the Registry

Some predefined structures and constants are widely used in COM/DCOM. Understanding them will greatly help to understand the COM/DCOM architecture. Besides, the Windows registry of the computers plays an important role in COM/DCOM implementation.

3.8.1 HRESULT

Most of COM/DCOM interface functions use HRESULT as their return value. This is because any function call can fail and COM/DCOM is designed to apply not only on a local machine, but also on networks(DCOM). Since network transmissions can fail, and the error conditions have to be reported, therefore, most of COM/DCOM functions will reserve the return value for reporting function status purpose. HRESULT is the data type to be returned to indicate the status of the function calls. By the same token, when implementing COM/DCOM interface functions, the functions should almost always return values of HRESULTs.

HRESULT is a 32-bit value. The WINERROR.H file contains the definitions of all of the COM/DCOM, OLE, ActiveX status codes currently generated by the system. By convention, the successful return codes contain an S_ in their name, while failure codes have an E_ in their names.

The most common HRESULT return values are:

S_OK	*Function succeeded.*
S_FALSE	*Function succeeded and returns a Boolean false.*
NOERROR	*Same as S_OK.*
E_UNEXPECTED	Unexpected failure.
E_NOTIMPL	Member function is not implemented.
E_NOINTERFACE	Component does not support requested interface.
E_OUTOFMEMORY	Could not allocate required memory.
E_FAIL	Unspecified failure.

One important point on using HRESULT return values is that the value can have multiple success codes or multiple failure codes to provide more information on the call status. This is why the SUCCEEDED and FAILED macros should be used to test if the HRESULT value is succeeded or failed. One should never compare the HRESULT value with S_OK or E_FAIL or other constants directly.

3.8.2 GUID

In COM/DCOM, every object such as a component or an interface which needs identified uniquely has an id associated with it. This id is defined as a GUID. A GUID is a 128-bit structure which stands for Globally Unique Identifier.

GUID is defined as:

```
typedef struct _GUID {
  unsigned long Data1;
  unsigned short Data2;
  unsigned short Data3;
  unsigned char Data4[8];
}
```

A GUID must be unique both over space and time. The GUID will uniquely identify the computer on that generates it. The GUID will also uniquely identify the time stamp when the GUID is created. To get a GUID, the Microsoft's utility program GUIDGEN.EXE or UUIDGEN.EXE can be used.

When a GUID is used for an interface, its typedefed symbol IID is usually used instead. When a GUID is used to identify a COM/DCOM component, the typedefed CLSID from GUID is used.

3.8.3 The Windows Registry

The Windows operating system has a system database called Registry. All the necessary information about the system's hardware, software, configuration and the users are stored in the Registry. Any Windows-based applications can place their information in the Registry so that their clients can retrieve the information there. The Registry can be explored by using the utility programs such as REGEDIT32.EXE under WindowsNT or REGEDIT.EXE under Windows95/98. Registry is a very important database and if it is messed up, the computer may not work properly.

The registry is a hierarchy of elements. Each element is called a key. A key can include a set of subkeys, a set of named values, and/or one unnamed value which is the default value. Subkeys can have other subkeys and values, but values cannot have subkeys or other values.

For COM/DCOM, one branch of the Registry, HKEY_CLASSES_ROOT, is used. Under this key, there is a subkey called CLSID. Under the CLSID subkey, the CLSIDs for all components installed on the computer are listed. Each CLSID key has a default value of string for its component's friendly name as CLSID itself is too complicated to look up. Each CLSID key has a subkey called InProcServer32 storing in-proc server information. The default value of this InProcServer32 subkey is the full path name of the DLL containing this component.

The CLSIDs and the file names are the most important information in the Registry for COM/DCOM components. Any component must have at least these information registered in the Registry.

Under the HKEY_CLASSES_ROOT key, there are also other subkeys that map a GUID to some piece of information. These subkeys include the following ones:

- AppID contains the subkeys used to map an APPID (application ID) to a remote server name. This key is used by DCOM.
- Component Categories maps the CATIDs(component category IDs) to a component category.
- Interface key is used to map IIDs to information specific to an interface.
- Licenses key stores licenses information on using COM/DCOM components.
- TypeLib key maps a LIBID to the file name where the type library is stored.

The registry also contains the ProgIDs which stands for programmatic identifiers under the HKEY_CLASSES_ROOT key. A ProgID maps a programmer-friendly name string

to a CLSID. Some languages such as Visual Basic, identify components by ProgID instead of CLSID. By convention, a ProgID has the following format:

ProgramNmae.ComponentName.Version

Once a component is registered in the Registry, some COM/DCOM library APIs can be used to query these information. The APIs CLSIDFromProgID and ProgIDFromCLSID can be used to find a CLSID from a ProgID or to find a ProgID from a CLSID.

Windows provides some APIs for Registry operations. The following ones might be useful when writing component registration code:

RegOpenKeyEx
RegCreateKeyEx
RegSetValueEx
RegEnumKeyEx
RegDeleteKey
RegCloseKey

3.9 In-proc Server

If a component is used in the same address space as its clients reside, the component is called an in-proc server. Let's look at how to implement a COM/DCOM in-proc server in this section.

Because some common tasks must be performed on all COM/DCOM components and clients, COM/DCOM defines a library of functions to make these tasks performed easier and in a standard and compatible way. The library is implemented in OLE32.dll.

3.9.1 Class Factory

First of all, a COM/DCOM component must be created to use by its clients. The COM/DCOM library provides a function, CoCreateInstance, for this purpose. This function is declared as:

```
HRESULT __stdcall CoCreateInstance(
  const CLSID &clsid,
  IUnknown *pIUnknown,
  DWORD dwClsContext,
  const IID &iid,
  void **ppv);
```

where clsid is the CLSID of the component to be created, pIUnknown is used to aggregate components, dwClsContext defines the component execution context, iid is the IID of the interface that is requested, ppv is the returned pointer to the requested interface;

The parameter dwClsContext represents the class context where the created component will be executed. This parameter can be a combination of the following values[3]:

CLSCTX_INPROC_SERVER means that the client will use the components in the same process, which implies that the components must be implemented in DLLs.

CLSCTX_INPROC_HANDLER means that the client will use in-proc handlers. An in-proc handler is an in-proc component that only implements part of the component and other parts are implemented in an out-of-process component.

CLSCTX_LOCAL_SERVER means that the client will use components in a different process but on the same computer.

CLSCTX_REMOTE_SERVER means that the client will use components on different machine across the network. This option requires DCOM to work.

There are other values defined as the combinations of the above values.

In-proc servers are faster because the function calls do not go across the process boundary. Out-of-process servers are secure as the components cannot access the clients' memory. Regardless of the class context, a component should behave the same way.

The function CoCreateInstance does not create the COM/DCOM component directly. Internally, it will create a class factory and use it to create the requested component. A class factory is a component used for creating other components. The reason to use the class factory is to isolate the knowledge on how to create a component from the clients of the component. The clients use the standard interface of the class factory, IClassFactory, through CoCreateInstance function to create the component.

The IClassFactory interface is declared as:

```
interface IClassFactory : IUnknown
{
  HRESULT __stdcall CreateInstance(IUnknown *pUnknownOuter,
                     const IID &iid,
                     void **ppv);
  HRESULT __stdcall LockServer(BOOL bLock);
};
```

The class factory is normally implemented along with the component since it will be used later by the clients of the component to create this component. That is, the CreateInstance

and LockServer functions have to be implemented for the component. The CreateInstance function will create the specified component. The function LockServer is used to manage the lifetime of the DLL server containing the component. When a client needs the DLL server, it can call LockServer(TRUE) and when it finishes with the server, it can call LockServer(FALSE) to release the server.

The function used to create a class factory for a specified component represented by its CLSID is CoGetClassObject. This function is called by the CoCreateInstance function. CoGetClassObject is defined as:

```
HRESULT __stdcall CoGetClassObject(
  const CLASID &clsid,
  DWORD dwClsContext,
  COSERVERINFO *pServerInfo,
  const IID &iid
  void **ppv);
```

pServerInfo is used by DCOM to control accessing remote components. The returned IID in CoGetClassObject and the CoCreateInstance functions are different. The returned IID in CoGetClassObject is the pointer to the class factory of the requested component, while in CoCreateInstance, the returned IID is the pointer to the requested component itself. The returned pointer from CoGetClassObject is usually an IClassFactory pointer and the desired component is created using this interface pointer.

Each instance of a class factory only creates components corresponding to a single CLSID. The class factory for a particular component is usually contained in the same DLL as the component it will create.

3.9.2 In-proc Server DLL Entry Points

For a client of an in-proc server, the COM library function CoGetClassObject needs an entry point in the DLL to create the component's class factory. This entry point function is specified as DllGetClassObject.

DllGetClassObject is declared as:

```
STDAPI DllGetClassObject(
  const CLSID &clsid,
  const IID &iid,
  void **ppv);
```

The STDAPI is a macro defined in OBJBASE.H which expands to

extern "C" HRESULT __stdcall

All COM/DCOM components must be registered in the registry so that their clients can find them. The DLL server must implement an exported function that will be called to register the components contained in it. The standard function for doing this is defined as:

STDAPI DllRegisterServer();

The DLL should also implement an exported function to unregister the components contained in it. The standard function doing this is defined as:

STDAPI DllUnregisterServer();

To control the lifetime of the DLL, the DLL must also implement an exported function so that the clients of the DLL can query the DLL if it can be unloaded from the memory safely. This standard function is defined as:

STDAPI DllCanUnloadNow();

For better memory management. COM/DCOM library implements a function called CoFreeUnusedLibraries to free the resource. This function calls the DLL's DllCanUnloadNow function to determine if the DLL can be freed from the memory. The client application should periodically call this function to free the unused resources.

3.9.3 Register/Unregister Servers

To register or unregister components contained in a DLL in the Registry. the Windows system provided utility program. REGSVR32.EXE. can be used. To register a DLL. run the REGSVR32.EXE in the command prompt as:

REGSVR32 -c dllName

To unregister a DLL. run it like:

REGSVR32 -u dllName

This utility program will load the DLL using LoadLibrary into the memory. get the pointer to DllRegisterServer function and call it to register. or get the pointer to DllUnregisterServer function and call it to unregister.

3.9.4 Example of an In-proc Server Component

Let's look at how an in-proc server component is implemented and how a client is implemented to use the in-proc server component using the COM/DCOM library functions.

Appendix 9 and Appendix 10 show an example from Rogerson[3] of a COM/DCOM in-proc server component and a client to use the component.

The server is registered in the registry by using the utility program regsvr32. The real registration code is written in the component. Usually, the registration or unregistration is performed in the setup program that is shipped with the component product.

During the execution of the application, the client will call COM/DCOM library function CoCreateInstance to create the component and request the interface IX. The CoCreateInstance will call CoGetClassObject which will in turn look for the component in the registry. If it finds it, it will load the DLL and call DLL's DllGetClassObject function. DllGetClassObject will create the class factory for the component, CFactory, and requests the IClassFactory interface from the created class factory. This IClassFactory interface pointer is returned to the CoCreateInstance and is used to call its CreateInstance function to create the component and returns the IX interface. After getting the interface, CoCreateInstance releases the class factory and return the IX pointer to the client. The client can then use the interface to use the services provided by the component through QueryInterface function.

3.10 Containment and Aggregation

Component reuse is certainly desired to reduce the duplication of program codes. How can a component be reused by other components?

COM/DCOM does not support implementation inheritance. This is because implementation inheritance violates the principle of component-based programming. The implementation inheritance will couple the object with its base class object. If the base object changes, the derived object will have to be changed as well.

On the other hand, COM/DCOM supports interface inheritance which a class inherits the interfaces of its base class. This allows a component to be extended or specialized. In COM/DCOM, the component specialization is achieved by using containment and aggregation.

Containment and aggregation are techniques which one component uses another component. One component contains another component or aggregates it. In containment, the container component contains pointers to the interfaces of the contained component. When the container component is created by its client, it will create the contained component and requests the contained component's interface pointers. These interface pointers are stored in the container component. The container component can then uses the contained component's services through these interface pointers. The container component is acting as a client of the contained component. Some of the container component's interfaces are implemented using the interfaces of the contained component. The container component can also re-implement an interface supported by the contained component. It can just forward the calls to the contained component through the stored contained component's interface pointers, or specialize the interface by adding code before or after the code for the contained component. From the point of view

of the container component's clients, the contained component did not exist. All functions are provided by the container component.

Aggregation is a specialization of containment. In aggregation, the container component passes the contained component's interface pointer directly to the client. The client then calls the contained component's interface directly. By doing aggregation, the container component does not need to re-implement and forward all of the functions in an interface. On the other hand, the container component cannot specialize any of the functions in the interface. The key in aggregation is that the client must not know that it is acting to two different components. The container and contained component must be made to behave as a single component. This is achieved by properly implementing the QueryInterface.

3.11 Local Server

If a component is implemented in an EXE file, it will run in a different process as its client application. The component is then an out-of-process server. If the component server is physically located on the same machine as its client, the server is known as a local server.

Each process has a different address space. If the client and the component are in different processes, the client cannot access the memory associated with an interface of the component. Therefore, a way of inter-process communication must be established. More precisely[3],

- A process needs to be able to call a function in another process;

- A process must be able to pass data to another process;
- A client should use the same way to access in-proc or out-of-proc components.

3.11.1 RPCs

In COM/DCOM, the technique used for inter-process communication is the Remote Procedure Call(RPC). For a local server, COM/DCOM uses the Local Procedure Call(LPC) which is a means of inter-process communication on a single machine based on RPC technique. For a remote server, COM/DCOM uses the RPC technique to communicate across the network. The RPCs are implemented by the operating system that can call functions in any process.

3.11.2 Marshaling

The parameters must be passed to a function from the address space of the client to the component's address space. This is known as marshaling. If both client and the component are on the same machine, then the data needs to be copied to the other address space. If the client and the component are on different machines, then the data has to be put in a standard format to tackle with the differences between the machines.

COM/DCOM provides an interface, IMarshal, for marshaling a component. IMarshal interface is responsible for marshaling the parameters before calling functions and

unmarshaling the parameters after calling functions. The COM/DCOM library implements a standard version of IMarshal that can be used for most of the interfaces.

3.11.3 Proxy/Stub DLLs

In COM/DCOM, the communication from the client to a component in a different process is through a DLL that acts like the component. This DLL is called a proxy. The client calls a function in the proxy and the proxy does the marshaling and the RPC calls to the actual code of the out-of-proc component. A proxy must be a DLL as it needs to access the address space of the client so it can marshal the data passed to the interface functions of the component.

Similarly, the component has a DLL to unmarshal the data that is marshaled by the client's proxy DLL. This DLL for the component is called a stub. The stub DLLs will also marshal the data sent back from the component to the client.

In a word, the client and the component in different processes communicate each other through the proxy and stub DLLs. The client communicates with the proxy DLL, the proxy marshals the data and calls the stub DLL using RPCs, the stub DLL unmarshals the data and calls the correct interface function in the component with the unmarshed parameters. The returned parameters are marshaled by the stub DLL to the proxy DLL and unmarshaled back to the client.

Since for every out-of-process server, the proxy and stub code has to be implemented and the RPC calls have to be made to cross the process boundary, a means is desired to automate all these implementations. The people at the Open Software Foundation(OSF) defined an environment called Distributed Computing Environment(DCE). A language called Interface Definition Language(IDL) and other things like the GUID design and the RPC specification are defined as part of the DCE. The IDL language is used to facilitate the task of creating proxy and stub DLLs and making RPC calls. This language, IDL, has a syntax similar to that of C and C++. It describes the interfaces and data shared by the client and the component. When the interfaces and the data are described by IDL, the MIDL compiler will generate the C-source files for the Proxy and Stub DLLs. After compiling and linking these C files, a DLL that implements the proxy and stub will be generated.

3.11.4 IDL

Writing IDL files is for providing enough information on the functions and their parameters in the interfaces so that the RPC can be made and the parameters can be marshaled. Here is an example of an IDL file for specifying an interface[3]:

```
import "unknwn.idl" ;
// Interface IX attribute list and the body
[
	object,
	uuid(32bb8323-b41b-11cf-a6bb-0080c7b2d682),
	helpstring("IX Interface"),
	pointer_default(unique)
]
interface IX : IUnknown
{
	HRESULT FxStringIn([in, string] wchar_t* szIn);
```

```
    HRESULT FxStringOut([out, string] wchar_t** szOut);
};
```

In IDL, each interface is specified with an interface header and an interface body. The interface header contains the attributes for the interface. The interface header is delimited by the square brackets([]). In the above example interface header, we have a keyword object which means this interface is a COM/DCOM interface. The uuid attribute is the IID to identify the interface. The helpstring attribute is to put a help string into the type library that is used in OLE Automation. The attribute pointer_default will tell the compiler the default way to treat pointers. This also applies to the embedded pointers, pointers that are in structures, unions, and arrays. The pointer_default attribute has three options[4]:

Ref: Pointers are treated as references. They cannot be NULL. They will not change within a function. They cannot be aliased within the function.

Unique: Pointers can be NULL. They can also change within a function. But they cannot be aliased within the function.

Ptr: Pointers are equivalent to the C pointers. That is, the pointers can be aliased, they can be NULL and they can change.

These attribute values in the header are used by the compiler to optimize the proxy and stub code it generates.

IDL also contains the keywords in and out as the function parameter attributes. As the attribute name implies, an in parameter is an input-only parameter going into the

function. An out parameter is an output-only parameter used as a returned value from the function. All out parameters specified in IDL must have pointer type. A parameter can also be marked as both in and out. For example.

*HRESULT foo([in] int x. [in. out] int *y. [out] int *z);*

The string attribute keyword used for function parameters informs the compiler that the parameter is a null-terminated string. The compiler can then determine the length of the string by looking for the terminating null character. It is very important when writing IDL files to specify the size of the data so that it can be copied during marshaling. In COM/DCOM. the strings used are the Unicode characters. wchar_t. or its equivalent macro OLECHAR. The Unicode is a character-encoding standard to take into account different types of characters in different languages such as Asian languages.

Functions in interfaces marked with object attribute in IDL must return HRESULTs. Therefore. other types of return values have to use out parameters.

By convention. if a function returns a string in an out parameter. the function usually allocates the memory for the string using CoTaskMemAlloc function or its equivalent APIs in OLE and other technologies and the caller of the function is then responsible for deallocating the memory for the string using CoTaskMemFree or its equivalent APIs..

The import keyword in IDL is used to include definitions from other IDL files. For instance. the above example includes the IDL definition for the IUnknown interface. All the standard COM/DCOM. OLE and ActiveX interfaces are defined in IDL files.

Here is another example of IDL file[3] on interface specifications that passes arrays and structures between the client and the component:

```
// Interface IY
[
        object,
        uuid(32bb8324-b41b-11cf-a6bb-0080c7b2d682),
        helpstring("IY Interface"),
        pointer_default(unique)
]
interface IY : IUnknown
{
        HRESULT FyCount([out] long* sizeArray);
        HRESULT FyArrayIn([in] long sizeIn,
             [in, size_is(sizeIn)] long arrayIn[]);
        HRESULT FyArrayOut([out, in] long* psizeInOut,
             [out, size_is(*psizeInOut)] long arrayOut[]);
};
// Structure for interface IZ
typedef struct
{
        double x;
        double y;
        double z;
} Point3d;
// Interface IZ
[
        object,
        uuid(32bb8325-b41b-11cf-a6bb-0080c7b2d682),
        helpstring("IZ Interface"),
        pointer_default(unique)
]
interface IZ : IUnknown
{
        HRESULT FzStructIn([in] Point3d pt);
        HRESULT FzStructOut([out] Point3d* pt);
};
```

In the above example, the size_is modifier is used in the function FyArrayIn to inform the IDL compiler, MIDL, that sizeIn is the number of elements in the array argument.

arrayIn. The size_is modifier can only be used with in or in-out parameter. The purpose of using the size_is modifier is again to specify the size of data for parameter marshaling.

Structures can also be defined in the IDL file and used as the function parameters. The above example defines a struct Point3d and uses it in the IZ interface functions.

If a structure contains pointers, the compiler MIDL must know exactly what a pointer is pointing to so that the data that the pointer references can be marshaled at runtime. Therefore, void* should never be used for a parameter.

Suppose an IDL file is called foo.idl, it can be compiled by the MIDL compiler using the following command:

midl foo.idl

After compiling, the compiler will generate a C/C++ compatible header file, foo.h, containing all declarations for the interfaces described in the IDL file. The name of this generated header file can be changed by using /h switch. The compiler will also generate a C file, foo_I.c, which defines all the GUIDs used in the IDL file. This file name can be changed by using /iid switch. Another C file, foo_p.c, is generated which implements the proxy and stub code for the interfaces in the IDL file. The name of the file can be changed by using /proxy switch. Finally, a C file, dlldata.c, is generated used to implement the DLL containing the proxy and stub code. Again, the file name can be changed by using /dlldata switch. As we will see later, the compiler may also generate a type library if there is a library statement in the IDL file.

3.12 Remote Server

Both local server and the remote server are out-of-process servers. However, a remote server is located physically on another computer in the network. Because of this, the communications between the client and the component are performed over the network.

There are two ways to use a remote server. The first way is to use a local server as a remote server by changing the Registry settings and the second way is to use an out-of-process server explicitly as a remote server.

3.12.1 Use Local Server Remotely

When a client is set up to use a local server, without changing the code of the client or the component, they can work each other across a network. To set this up, two systems are required to be connected over a network, each is running either Windows NT4.0 or Windows 95/98 with DCOM installed. To let the client use the remote server, a DCOM configuration tool, DCOMCNFG.EXE, can be employed. This tool allows the user to change various parameters for the applications on the computer including whether they run locally or remotely.

The following steps show the procedure to run a server remotely[3]:

1. Build the client executable, the server executable and the proxy dll on the local machine;

2. Copy the client executable, the server executable and the proxy dll to the remote machine;
3. Register the server executable as a local server on both the local machine and the remote machine;
4. Run the client executable to use the local server on both the local and the remote machines to make sure that the programs are working on both machines;
5. Run DCOMCNFG.EXE on the local machine. Select the component, and click Properties. Select the Location tab. Deselect the option Run Application On This Computer, and select the option Run Application On The Following Computer. Type the name of the remote computer. Click the Identity tab, and select the Interactive User radio button;
6. Run the server executable on the remote machine;
7. Run the client executable on the local machine;
8. Verify that both client and the server executable are working correctly.

When the local server is registered, The CLSID subkey under the HKEY_CLASSES_ROOT key will store the information about the component. It will have the default value which is the friendly name for the component. It will also store the path to the application in which the component is implemented under the LocalServer32 subkey.

When running DCOMCNFG.EXE for the component, this program will add an AppID named value under the CLSID subkey for the local server component in the Registry and an AppID subkey under the HKEY_CLASSES_ROOT key. An AppID is a GUID. It will

store the information used by DCOM. The AppID subkey will have at least three values. The default value is a friendly name. The named value RemoteServerName contains the name of the server where the application is located, the named value RunAs tells the DCOM how to run the application. In addition, the name of the application is stored directly under the AppID key.

In DOCM, the CoGetClassObject function will open the specified component server with the correct context. If the context is CLSCTX_LOCAL_SERVER, CoGetClassObject will check the Registry to see whether the specified component has an AppID. If it does, the function then checks the RemoteServerName key in the Registry. If it finds the server name, CoGetClassObject then attempts to run the server remotely. This is the mechanism why a local server can be used as a remote server in DCOM.

3.12.2 Use Remote Server Explicitly

A remote server can also be programmatically specified by the client to be accessed remotely. To use this method, the client must use the DCOM function CoCreateInstanceEx or modifying the calls to CoGetClassObject.

The following example code[3] demonstrates how to use CoCreateInstanceEx to create a remote server component.

```
// fill in the server information
COSERVERINFO ServerInfo;
memset(&ServerInfo, 0, sizeof(ServerInfo));
ServerInfo.pwszName= L"ExampleRemoteServer";
```

```
// set up MULTI_QI structure with the desired interfaces
MULTI_QI mqi[3]:
mqi[0].pIID= IID_IX: //[in] IID of desired interface
mqi[0].pItf= NULL:  //[out] pointer to interface
mqi[0].hr= S_OK:    //[out] result of query interface

mqi[1].pIID= IID_IY:
mqi[1].pItf= NULL:
mqi[1].hr= S_OK:

mqi[2].pIID= IID_IZ:
mqi[2].pItf= NULL:
mqi[2].hr= S_OK:

HRESULT hr=
  CoCreateInstanceEx(CLSID_Component1.
            NULL.
            CLSCTX_REMOTE_SERVER.
            &ServerInfo.
            3. //number of interfaces to request
            &mqi):
```

Calling CoCreateInstanceEx. the remote server information can be specified directly by the COSERVERINFO parameter. This function also allows query of multiple interfaces at the same time by using the MULTI_QI parameter. This will save the network traffic overhead.

4 OLE Automation

Microsoft's Object Linking and Embedding(OLE) is an important technology on component-based programming. We will overview the OLE functionality and discuss OLE Automation technique.

4.1 What is OLE?

OLE is a mechanism for communications between components. It provides a standard conceptual framework to create, manage, and access various components that provide services to other components or clients. OLE is a technology based on COM/DCOM to allow applications to transfer and share information.

OLE originally stands for "Object Linking and Embedding" when it was designed for making compound document, i.e., to place a reference of a document into another document or physically copy a document into another document to make the latter a compound document, a document containing other documents. Today's OLE is not just as its acronym stands for. It has become the term that represents the application component communication technology.

Built on top of COM/DCOM, OLE has the following core facilities[5]:

1. The structured storage model defines a scheme to store and retrieve objects that reside inside files and other containers.
2. The moniker mechanism that provides persistent naming of objects.

3. The uniform data transfer model provides a single, standard mechanism for transfer objects and data between applications.

OLE provides a solution to store multiple types of objects in one document, namely a file system within a file. A single file is treated as a collection of two types of objects: storages and streams. Storages act as directories and streams act as files. Together, they function as a file system inside a file. An OLE compound file has a root storage object containing at least one stream object for its native data and some storage objects corresponding to its linked and embedded objects. Therefore, a compound file can have many levels of nested objects.

The monikers provide a way to identify an object. The monikers are themselves objects which provide services for binding, allowing a component to obtain a pointer to the object identified by the moniker.

The OLE uniform data transfer model provides a mechanism for transferring data between applications through clipboard, through compound document or by using drag and drop.

Based on these core facilities, OLE currently provides two features: Automation and Compound Document.

Automation provides another way for a client to communicate with a component. This technique will be discussed in the rest of this section.

4.2 Automation

In COM/DCOM, a component implements the functions of some interfaces. The client code includes a header file containing the declarations of the interfaces. If a client of the component wants to use the component's services, it gets the interface pointer and calls the interface functions directly.

OLE Automation, or Automation for short, provides another mechanism of communications between a component and its clients. In Automation, the component offers its services through a single standard interface called IDispatch. It implements the functions of this standard interface. The client of the component calls the functions of this standard interface to use the component's services.

The COM/DCOM component in Automation is called an Automation server and the client of the Automation server is called an Automation controller. An Automation controller does not need to include the header files containing the declarations of the server's interfaces. Instead, it passes the function name and the parameters required by the function to the IDispatch interface function, and based on these runtime information provided by the controller, the Automation server calls its right function. Hence, Automation does not perform compile-time type checking, also known as early-binding. It does run-time type checking, also known as late-binding.

The intention for Microsoft to invent this Automation technique is to make it easier for interpretive and macro languages, such as Visual Basic, Java, Microsoft Word and Microsoft Excel, to access COM/DCOM components. Automation will also make it

easier to write components using these languages since Automation provides the same way to execute a function via the function's name as these languages' run-time system does.

4.3 IDispatch

IDispatch is the single standard interface in Automation for the controllers to access the servers. The IDispatch interface in IDL format is defined in Oaidl.idl as follows:

```
interface IDispatch : IUnknown
{
  HRESULT GetTypeInfoCount([out] UINT *pctinfo);
  HRESULT GetTypeInfo([in] UINT itinfo,
              [in] LCID lcid,
              [out] ITypeInfo ** pptinfo);
  HRESULT GetIDsOfNames([in] REFIID riid,
               [in, size_is(cNames)] LPOLESTR *rgszNames,
               [in] UINT cNames,
               [in] LCID lcid,
               [in, out, size_is(cNames)] DISPID *rgdispid);
  HRESULT Invoke([in] DISPID dispidMember,
          [in] REFIID riid,
          [in] LCID lcid,
          [in] WORD wFlags,
          [in, unique] DISPPARAMS *pdispparams,
          [in, out, unique] VARIANT *pvarResult,
          [out] EXCEPINFO *pexcepinfo,
          [out] UINT *puArgErr);
};
```

IDispatch interface, like any COM/DCOM interface, is derived form IUnknown. Therefore, any Automation server must implement the three functions in IUnknown, that is, the QueryInterface, AddRef and Release functions, and the above four functions in IDispatch.

In OLE, type information is a term referring to any information about objects and their interfaces. A C/C++ header file tells all the information about its implementation file, such as what functions are declared in the file, what parameters each of these functions have, and what data is defined in this file, etc. The type information can contain all the necessary information like what interfaces an object supports, what functions each of these interfaces contains, what parameters each of these functions requires and so on.

GetTypeInfoCount function returns the number of type descriptions for the object. For Automation objects that support IDispatch, the type information count is 1. If an object does not provide type information, this function should return 0. GetTypeInfo retrieves the type information for an object.

The two most interesting functions in IDispatch are GetIDsOfNames and Invoke. In Automation, each name such as a function name or an argument name is associated with a long integer ID. This ID is called a dispatch ID, also known as DISPID. The DISPID must be unique within an implementation of IDispatch. Each implementation of IDispatch, as an interface, is identified by its own IID. The function GetIDsOfNames maps a single member function and an optional set of argument names to a corresponding set of DISPIDs, which can then be used to call the IDispatch::Invoke function.

The parameter riid in GetIDsOfNames is reserved for future use and it must be IID_NULL currently. The parameter rgszNames in the function is the passed-in array of names to be mapped. Parameter cNames is the count of the names to be mapped. Parameter lcid is the locale context in which to interpret the names. A locale represents a

language for a particular geographical region. For example, Canada has two languages officially: English and French. Thus, Canada has two distinct locales: Canadian-English and Canadian-French. Therefore, lcid specifies if the names are in U.S. English or Canadian French, or in another language. Parameter rgdispid is the returned array, each element of which contains a DISPID corresponding to one of the names passed in the rgszNames array. The first element represents the function name and the subsequent elements represent each of the function's parameters. A function in Automation is usually referred to as a method.

Zero and negative IDs are reserved regarding DISPIDs. Therefore, an IDispatch implementation can only use positive DISPID value with a given name. The method and method's parameter DISPIDs should be kept unchanged to allows a controller to obtain these DISPIDs once, and cache them for later use.

Whenever an Automation controller wants to access a server's exposed properties or methods, it must call the server's Invoke function. To execute a method, the controller passes the DISPID to the Invoke function of the IDispatch interface and based on the DISPID of the method, the server will find and call the correct function. That is, the behavior of the Invoke function is controlled by the DISPID passed to it and the DISPID is used as an index for the methods supported in the dispatch interface.

The first parameter of the Invoke function, dispidMember, is the DISPID of the member function to call. The DISPID can be retrieved by using GetIDsOfNames or from the

object's documentation. The second one is reserved and must be IID_NULL. The third parameter is the locale information. The rest parameters will be explained subsequently.

4.4 Dispinterface

An implementation of the IDispatch's Invoke function consists of a set of functions. These functions define the services that the Automation server supports. In Automation, this set of the functions implemented by the Invoke function is called a dispatch interface or dispinterface for short.

4.5 Dual Interface

A COM/DCOM interface provides functions which can be called by its clients through the function pointers in the virtual table, or vtbl. A dispinterface, however, makes its functions available through the IDispatch::Invoke method. If an interface makes its functions available both through the vtbl and through the dispinterface, it is called a dual interface. Therefore, a dual interface is a COM/DCOM interface that inherits from IDispatch. Dual interface implementation is preferred because it allows C++ programs to make their calls through vtbl which is faster to execute, while macros or interpretive languages still can use the interface through Invoke method.

4.6 Using IDispatch interface

When an Automation controller uses a dispinterface, the controller has to know the ProgID of the Automation server and the function name. The controller will use the

ProgID to get the CLSID of the server by calling CLSIDFromProgID API. The CLSID can then be used to create the server and get the server's IDispatch interface. This interface is then used to get the DISPID for the function by calling GetIDsOfNames member function of the IDispatch interface. Once this DISPID is available, the function can be called indirectly by calling the Invoke function.

The following is an example[3] on how to use IDispatch interface in an Automation controller's code.

```
// initialize the OLE library
HRESULT hr= OleInitialize(NULL);

// get the CLSID of the component
// OLE uses wide character strings
wchar_t progID[]= L"Example Dispatch Component";
CLSID clsid;
::CLSIDFromProgID(progID, &clsid);

// create the component and get the IDispatch interface
IDispatch *pIdispatch= NULL;
::CoCreateInstance(clsid, NULL,, CLSCTX_INPROC_SERVER,
        IID_IDispatch, (void** )&pIdispatch);

// get the DISPID of the function we want
DISPID dispid;
OLECHAR *name= L"Fx";
pIDispatch->GetIdsOfNames(
 IID_NULL, //reserved, must be IID_NULL
 &name,  //the function name
 1,      //number of names
 GetUserDefaultLCID(), //locale info
 &dispid); //return the DISPID of Fx

// prepare to call Fx. Named arguments will make their
// orders immaterial to the IDispatch implementation.
DISPPARAM dispParam=
 {NULL, //array of VARIANT arguments
 NULL, //array of dispatch IDs of named arguments
 0,  //number of arguments
 0}; //number of named arguments
```

```
// call Fx indirectly through Invoke
pIDispatch->Invoke(
  dispid,              //DISPID of Fx function
  IID_NULL,            //reserved, must be IID_NULL
  GetUserDefaultLCID(),//locale info
  DISPATCH_METHOD,     //normal method
  &dispParam,          //method arguments
  NULL,                //results
  NULL,                //exception info
  NULL);               //arg error
```

Note that in COM/DCOM and OLE, the strings used are wide character strings. This is why any string literal should be prefixed with L as defined in C++.

4.7 Methods and Properties

A C++ object has methods and variables. Variables define the properties and methods act on these variables. In COM/DCOM, all members of an interface are functions. There are no variables for interfaces. How to implement properties in interfaces? One way is to use the "Get/Set" functions. In IDL, the propget and propput attributes specify that an interface function is treated as a property. The following example interface definition[3] shows how to define a property in IDL:

```
interface IWindow : IDispatch
{
  [propput]
  HRESULT Visible([in] VARIANT_BOOL bVisible);
  [propget]
  HRESULT Visible([out, retval] VARIANT_BOOL *pbVisible);
}
```

This example interface definition segment specifies that the interface has a property called Visible. The function marked with propput takes a parameter as a value to set the

property Visible and the function marked with propget returns the value for property Visible. The name of the property is the same as the name of the function, i.e., Visible. When the IDL is compiled by MIDL, the compiler will attach the prefix get_ or put_ to the functions based on the attribute associated with the function name. VARIANT_BOOL used in the above example is a type used in OLE for Boolean variables, and its value VARIANT_FALSE is 0 and VARIANT_TRUE is defined as 0xFFFF.

The retval attribute in IDL designates the parameter that receives the return value of the member function. This attribute can be used only on the last parameter of the member. The parameter must have the out attribute and must be a pointer type.

Because the way of property is implemented in Automation, a single name for a DISPID might be associated with four different functions: a normal function, a function to put a property, a function to get a property and a function to put a property by reference. They all have the same function name and the same DISPID. Therefore, we need a way to distinguish them. The fourth parameter in IDispatch's Invoke function is a flag used for this purpose. It has the following four constant values:

DISPATCH_METHOD
DISPATCH_PROPERTYGET
DISPATCH_PROPERTYPUT
DISPATCH_PROPERTYPUTREF

4.8 VARIANTs

Since IDispatch::Invoke is a generic function to call any function in the dispinterface which could take any number of arguments, each argument could be any type of data, the arguments must be supplied in a generic way. The fifth parameter in the Invoke function takes the parameters to the function being invoked. It is a pointer to a structure used in Automation called DISPPARAMS which is defined as follows:

```
typedef struct tagDISPPARAMS{
  VARIANTARG FAR* rgvarg;
  DISPID FAR* rgdispidNamedArgs;// Dispatch IDs of named arguments
  unsigned int cArgs;          // Number of arguments
  unsigned int cNamedArgs;     // Number of named arguments
} DISPPARAMS;
```

The rgvarg field is an array of arguments, each has a VARIANTARG type. The field rgdispidNamedArgs is an array of DISPIDs of named arguments which is used in Visual Basic. Named arguments allow the parameters to be passed to a function in any order by specifying the names of the parameters because the arguments will be identified by their names. This concept is not used in C++. The field cArgs denotes the number of arguments in the cArgs array and cNamedArgs represents the number of named arguments.

The VARINATARG type used in the DISPPARAMS structure is the same as VARIANT. VARIANT is an important generic data type used in Automation. A VARIANT is a union of different types. So it can represent all types of data that specified in its union fields. All data must be converted to VARIANTs before passing in dispinterfaces.

Here is the definition of the VARIANT type:

```
struct tagVARIANT{
  VARTYPE vt;
  WORD wReserved1;
  WORD wReserved2;
  WORD wReserved3;
  union
  {
    long          lVal;                 /* VT_I4                    */
    unsigned char bVal;                 /* VT_UI1                   */
    short         iVal;                 /* VT_I2                    */
    float         fltVal;               /* VT_R4                    */
    double        dblVal;               /* VT_R8                    */
    VARIANT_BOOL  boolVal;              /* VT_BOOL                  */
    SCODE         scode;                /* VT_ERROR                 */
    CY            cyVal;                /* VT_CY                    */
    DATE          date;                 /* VT_DATE                  */
    BSTR          bstrVal;              /* VT_BSTR                  */
    IUnknown      *punkVal;             /* VT_UNKNOWN               */
    IDispatch     *pdispVal;            /* VT_DISPATCH              */
    SAFEARRAY     *parray;              /* VT_ARRAY|*               */
    unsigned char *pbVal;               /* VT_BYREF|VT_UI1          */
    short         *piVal;               /* VT_BYREF|VT_I2           */
    long          *plVal;               /* VT_BYREF|VT_I4           */
    float         *pfltVal;             /* VT_BYREF|VT_R4           */
    double        *pdblVal;             /* VT_BYREF|VT_R8           */
    VARIANT_BOOL  *pbool;               /* VT_BYREF|VT_BOOL         */
    SCODE         *pscode;              /* VT_BYREF|VT_ERROR        */
    CY            *pcyVal;              /* VT_BYREF|VT_CY           */
    DATE          *pdate;               /* VT_BYREF|VT_DATE         */
    BSTR          *pbstrVal;            /* VT_BYREF|VT_BSTR         */
    IUnknown      **ppunkVal;           /* VT_BYREF|VT_UNKNOWN      */
    IDispatch     **ppdispVal;          /* VT_BYREF|VT_DISPATCH     */
    SAFEARRAY     **pparray;            /* VT_BYREF|VT_ARRAY|*      */
    VARIANT       *pvarVal;             /* VT_BYREF|VT_VARIANT      */
    void          * byref;              /* Generic ByRef            */
  };
};
typedef struct tagVARIANT VARIANT;
```

The vt field of the VARIANT indicates the type of the data stored in the VARIANT. For example, if vt is VT_BSTR, then we can only use the bstrVal field as the data.

Due to the characteristics of run-time binding of the data type in dispinterface, only a generic type of data can be specified for the functions' arguments. Unlike using a C++ header file to strictly specify the argument types at compile time, a VARIANT can be used to let the Automation server to bind the data type at run-time.

For the Invoke function in IDispatch interface, the sixth parameter, pVarResult, is a pointer to a VARIANT that holds the result of the method or the propget executed through Invoke function. If the method has no return value or for propput and propputref, then this parameter is NULL.

The seventh parameter of the Invoke function is used for exception handling purpose which is a pointer to an EXCEPTINFO structure that should be filled with the exception information if an error occurs on executing the method or the property.

The last parameter in Invoke function, puArgErr, will contain the index of the argument corresponding to a DISP_E_PARAMNOTFOUND or DISP_E_TYPEMISMATCH error if such errors occurred.

4.9 BSTRs

One VARIANT type is BSTR which is widely used for strings. A BSTR is a pointer to a wide character string. There are three characteristics for the BSTRs[3]:

1. A BSTR is a counted string.
2. A BSTR stores the count before the array of characters.

3. A BSTR can have multiple null characters inside the string.

Due to these characteristics, a string cannot be assigned to a BSTR directly like:

```
BSTR bStr= L"Test BSTR";
```

because the count will not get set correctly. You have to use a Win32 API. SysAllocString, to allocate a BSTR. For example,

```
wchar_t wstr[]= L"Test BSTR";
BSTR bStr= SysAllocString(wstr);
```

Once a BSTR is allocated, the Win32 API SysFreeString should be used to deallocate the BSTR.

4.10 SAFEARRAYs

Another VARIANT type is worth of paying attention to, the SAFEARRAY. A SAFEARRAY is an array that includes the bound information. The SAFEARRAY is defined as follows:

```
typedef struct tagSAFEARRAYBOUND {
   ULONG cElements;
   LONG lLbound;
} SAFEARRAYBOUND;

typedef struct tagSAFEARRAY {
   unsigned short cDims;
   unsigned short fFeatures;
   unsigned long cbElements;
   unsigned long cLocks;
   BYTE *pvData;
   [size_is(cDims)] SAFEARRAYBOUND rgsabound[];
} SAFEARRAY;
```

The structure SAFEARRAYBOUND represents the bounds of one dimension of the array. The lower bound of the dimension is represented by lLbound field. and cElements represents the number of elements in the dimension.

In the SAFEARRAY structure. the cDims field is the number of the dimensions of the array. The fFeatures specifies what type of data is stored in the SAFEARRAY and how the array is allocated. The allowed data types in an SAFEARRAY are as follows:

FADF_BSTR An array of BSTRs
*FADF_UNKNOWN An array of IUnknown**
*FADF_DISPATCH An array of IDispatch**
FADF_VARIANT An array of VARIANTs

The ways of allocations for the SAFEARRAYs are listed as well:

FADF_AOTU The array is allocated on the stack
FADF_STATIC The array is statically allocated
FADF_EMBEDDED The array is embedded in a structure
FADF_FIXEDSIZE The array cannot be resized or reallocated

The Automation library contained in OLEAUT32.DLL contains a set of functions to manipulate SAFEARRAYs. All these functions begin with the prefix SafeArray.

4.11 Type Libraries

So a controller can access a server through a dispinterface without any type information about the dispinterface or its method. It can use the run-time type checking and conversion on VARIANTs.

However, this kind of type checking and conversion is very time-consuming and error prone. Therefore, we need a language independent equivalent of the C++ header files that is suitable for interpretive and macro languages to get the server's exposed objects information. In Automation, type libraries are used for such purpose to specify type information about the servers, their interfaces, methods, properties, arguments and structures. The type libraries act like the C++ header files.

A type library is a binary file generated by the compiler, MIDL, from the source code written in IDL. The type libraries will be accessed programmatically and the OLE library provides standard components for creating and reading the type libraries. With the help of the Type Libraries, Visual Basic can access a server through the vtbl part of its dual interface, which is faster and type safe. Otherwise, Visual Basic can only access the server through its dispinterfaces.

Here an example of an IDL file which can be compiled into a type library[3]:

```
// Interface IX header and body
[
	object,
	uuid(32BB8326-B41B-11CF-A6BB-0080C7B2D682),
	helpstring("IX Interface"),
	pointer_default(unique),
	dual,
	oleautomation
]
interface IX : IDispatch
{
	import "oaidl.idl";
	HRESULT Fx();
	HRESULT FxStringIn([in] BSTR bstrIn);
	HRESULT FxStringOut([out, retval] BSTR* pbstrOut);
	HRESULT FxFakeError();
};
```

```
// type library header and body
[
	uuid(D3011EE1-B997-11CF-A6BB-0080C7B2D682),
	version(1.0),
	helpstring("Example Type Library")
]
library ServerLib
{
	importlib("stdole32.tlb");
	// Component header and body
	[
		uuid(0C092C2C-882C-11CF-A6BB-0080C7B2D682),
		helpstring("Component Class")
	]
	coclass ExampleComponent
	{
		[default] interface IX;
	};
};
```

The dual attribute identifies an interface as a dual interface that exposes properties and methods through IDispatch and also directly through the vtbl. The oleautomation attribute indicates that an interface is compatible with Automation. The keyword library indicates that everything inside the library block will be compiled into a type library. After compiling by the MIDL compiler, a type library with the given name, e.g., ServerLib.tlb for the above example, will be generated. The importlib directive will include type library content from other existing type library. The version attribute identifies a particular version of an interface. The coclass keyword defines a component. In the above example, it defines the component with name ExampleComponent that has a single interface IX.

Once the type libraries are generated, they can be shipped as separate files or included in the resource file of the application. To use a type library, it must be loaded in the memory. We can call LoadRegTypeLib if the type library is stored in the Registry. We

can also call LoadTypeLib to load the type library from a disk file. Or call LoadTypeLibFromResource to load it from the resource in an EXE or a DLL. If the type library is successfully loaded, an ITypeLib interface pointer will be returned. This pointer can then be used to register the type library by calling RegisterTypeLib API. The type library can now be used to get the interfaces or components by calling ITypeLib::GetTypeInfoOfGuid function passing the CLSID or IID.

The following is an example on using the type libraries[3].

```
HRESULT hr;
// Load TypeInfo on demand if we haven't already loaded it.
if (m_pITypeInfo == NULL)
{
 ITypeLib* pITypeLib = NULL;
 hr = ::LoadRegTypeLib(LIBID_ServerLib,
                       1, 0, // version numbers
                       0x00,
                       &pITypeLib);
 if (FAILED(hr))
 {
  hr = ::LoadTypeLib("Server.lib",
                          &pITypeLib);
  if(FAILED(hr))
  {
    trace("LoadTypeLib Failed.", hr);
    return hr;
  }

  // Ensure that the type library is registered.
  hr = RegisterTypeLib(pITypeLib, wszTypeLibFullName, NULL);
  if(FAILED(hr))
  {
    trace("RegisterTypeLib Failed.", hr);
    return hr;
  }
}

// Get type information for the interface of the object.
hr = pITypeLib->GetTypeInfoOfGuid(IID_IX,
                                  &m_pITypeInfo);
```

```
  pITypeLib->Release();
  if(FAILED(hr))
  {
    trace("GetTypeInfoOfGuid failed.", hr);
    return hr;
  }
}
return S_OK;
```

The type library must be registered in the Registry before it can be used by Automation controllers. After the type library is registered, its LIBID, a GUID, will be written under the key HKEY_CLASSES_ROOT\TypeLib.

4.12 Implementing IDispatch

There are various ways to implement IDispatch interface. For instance, a table of function names and function pointers can be built for the dispinterface. However, the simplest and most popular way to implement IDispatch is to delegate GetIDsOfNames and Invoke to the ITypeInfo interface pointer for the implemented interface.

As shown before, after the type library is loaded, a pointer to the ITypeLib will be received. The ITypeInfo interface pointer can be retrieved by calling ITypeLib::GetTypeInfoOfGuid passing it the IID of the interface.

The following is an example of IDispatch implementation[3]:

```
HRESULT __stdcall CA::GetTypeInfoCount(UINT* pCountTypeInfo)
{
        *pCountTypeInfo= 1;
        return S_OK;
}
```

```
HRESULT __stdcall CA::GetTypeInfo(
        UINT iTypeInfo,
        LCID,       // This object does not support localization.
        ITypeInfo** ppITypeInfo)
{
        *ppITypeInfo= NULL;
        if(iTypeInfo != 0)
                return DISP_E_BADINDEX;

        // Call AddRef and return the pointer.
        m_pITypeInfo->AddRef();
        *ppITypeInfo= m_pITypeInfo;
        return S_OK;
}

HRESULT __stdcall CA::GetIDsOfNames(
        const IID& iid,
        OLECHAR** arrayNames,
        UINT countNames,
        LCID,       // Localization is not supported.
        DISPID* arrayDispIDs)
{
        if (iid != IID_NULL)
                return DISP_E_UNKNOWNINTERFACE;

        HRESULT hr= m_pITypeInfo->GetIDsOfNames(arrayNames,
                                countNames,
                                arrayDispIDs);
        return hr;
}

HRESULT __stdcall CA::Invoke(
    DISPID dispidMember,
    const IID& iid,
    LCID,       // Localization is not supported.
    WORD wFlags,
    DISPPARAMS* pDispParams,
    VARIANT* pvarResult,
    EXCEPINFO* pExcepInfo,
    UINT* pArgErr)
{
        if (iid != IID_NULL)
                return DISP_E_UNKNOWNINTERFACE;

        ::SetErrorInfo(0, NULL);
        HRESULT hr= m_pITypeInfo->Invoke(
```

```
            static_cast<IDispatch*>(this),
            dispidMember, wFlags, pDispParams,
            pvarResult, pExcepInfo, pArgErr);
      return hr;
}
```

4.13 Exception Handling

Exception handling is important in writing robust code. The mechanism for exception handling in Automation is as follows:

The Automation server must implement the interface ISupportErrorInfo which only has one function InterfaceSupportsErrorInfo. For example,

```
virtual HRESULT __stdcall InterfaceSupportsErrorInfo
  (const IID &iid)
{
  return (iid == IID_IX) ? S_OK : S_FLASE;
}
```

which tells that interface IID_IX support error reporting.

Then in the implementation of IDispatch::Invoke, call SetErrorInfo(0, NULL) before calling ITypeInfo::Invoke. This will set the error information object to NULL for the current thread of execution. The first parameter of SetErrorInfo is reserved and must be 0.

When an exception is thrown, call CreateErrorInfo to get an ICreateErrorInfo interface pointer. Use this pointer to fill in the information about the error. Call SetErrorInfo, passing it the ICreateErrorInfo pointer. The client should react to the exception.

Here is an example on implementing error reporting when an exception occurs[3]:

```
// Create the error info object.
ICreateErrorInfo* pICreateErr;
HRESULT hr = ::CreateErrorInfo(&pICreateErr);
if(FAILED(hr))
	return E_FAIL;

pICreateErr->SetSource(L"Example Component");
pICreateErr->SetDescription(
	L"An error generated by the component.");
IErrorInfo* pIErrorInfo = NULL;
hr = pICreateErr->QueryInterface(IID_IErrorInfo,
			(void**)&pIErrorInfo);
if(SUCCEEDED(hr))
{
	::SetErrorInfo(0L, pIErrorInfo);
	pIErrorInfo->Release();
}
pICreateErr->Release();
return E_FAIL;
```

5 ActiveX Controls

ActiveX controls apply the principles of COM/DCOM and OLE. In this section, we will discuss ActiveX controls in general and on how to create an ActiveX control using Microsoft Foundation Classes (MFC) in detail.

5.1 ActiveX Control Overview

A control is a user interface object. Controls are used to provide the building blocks for creating user interfaces in applications. A control container is the client of the control. A control is typically embedded in the control container. For example, a button control allows the user to click on it and this clicking event is passed to the control container to take some action. Other common controls include text boxes, radio buttons, check boxes, list boxes, tree controls, etc.

5.1.1 What is an ActiveX Control?

An ActiveX control is a control and it is also a COM/DCOM component. Since an ActiveX control is a COM/DCOM component, it exposes its IUnknown interface. Since an ActiveX control is used as a control, it is usually an in-proc server.

5.1.2 Characteristics of ActiveX Controls

Since an ActiveX control will be used to perform some user interface activities such as responding to mouse and keyboard events, it should have a visual representation. This is

common to all controls. As a control, it should also have the capability to notify its container about what have happened on it.

ActiveX controls can be placed on the web pages.

ActiveX controls can be dragged and dropped by using OLE Uniform Data Transfer facility.

An ActiveX control can also support in-place activation. In-place activation is a user interface technique. When a user activates an embedded server such as an ActiveX control within a container, a new menu bar that consists of both menu items from the container and from the activated server will replace the container's original menu bar. When a user deactivates the server, the container's original menu bar will be restored.

Each ActiveX control usually has some property pages associated with it to allow the user to view and change its properties.

ActiveX controls usually support OLE Automation..

As a control, an ActiveX control should save its states such as its property values and its user interface status.

5.1.3 Features of ActiveX Controls

Due to its characteristics. ActiveX controls usually need to provide some specific functionality.

Language Integration

ActiveX controls are programmable objects. The container should be able to communicate with an ActiveX control embedded in it using any programming language so that it can get or set the control's properties. call its methods and react to its notifications. To facilitate this task. ActiveX controls should support OLE Automation.

A client site is a means for the embedded object to request services from its container. A container can implement client sites for the ActiveX controls in it.

A container can also aggregate the ActiveX control to extend the controls functionality. Therefore. all ActiveX controls should be able to be aggregated. All controls created with MFC can be aggregated.

Ambient Properties

Ambient properties are the properties of the container and they can be used by its embedded objects such as ActiveX controls as their environmental values. The container can make the decision on what ambient properties it will implement. The embedded object can choose which of the ambient properties it uses. The ambient property values cannot be changed by the embedded objects. They can be changed by the user of the

container application. If the user changes an ambient property, the new property value will be applied to those embedded objects that are using the changed ambient property.

The ActiveX controls specification defines a list of standard ambient properties. Each standard ambient property has a name and is associated with a specific DISPID. The DISPID designates the ambient property.

The container's client site will expose the ambient properties through its IDispatch interface so that the embedded ActiveX controls can get them through this interface.

Events

The controls can send notifications to its container. These notifications are called events. The meaning of each event is defined by the control and the container should provide interface to react to these events. The control can fire events at any time after it is embedded in the container.

Since the control fires events and the container provides an interface to handle these events, the control is called an event source and the container's interface is called an event sink. The events are handled by the container through an Automation interface. The control's type library describes the event interface that it wants its container to handle. But the event interface is marked by the control as a source in the coclass section of its Object Description Language(ODL) file. This means that the control will not implement it.

IDL is a general language for specifying components, interfaces and other objects. ODL is a specific language for OLE and is extended from IDL. In terms of defining OLE interfaces, ODL and IDL are the same.

Here is an example from Denning[7] of the ODL file segment for event descriptions:

```
[
  uuid(37D341A5-6B82-101B-A4E3-08002B291EED),
  helpstring("example control")
]
coclass exampleControl
{
  [default] interface IExample;
  interface IDispatch;
  [default, source] interface IExampleEvents;
};
```

This example specifies that the component exampleControl will have three Automation interfaces, IExample, IDispatch, and the IexampleEvents interfaces. The IExample will be implemented by the control itself. However, IExampleEvents will not be implemented by the control because it is marked as source. The attribute default designates the interface as the default Automation interface.

How does the control connect to the container's Automation interface for its events? The answer is to use a technique called connection points.

Connection Points

A connection point is an interface exposed by an object. The responsibility of this interface is to hook up to the implementations of an interface in another object so that the former object can talk to the latter object. Take the ActiveX control's events as an

example. A control describes the event interface as an Automation interface in its type library. But it marks the interface as a source. The control then provides a connection point so that the container can connect its implementation of the control's event's Automation methods to the control.

A connection point is defined in the IConnectionPoint interface. It has the following methods:

*HRESULT GetConnectionInterface(IID *pIID);*
*HRESULT GetConnectionPointContainer(IConnectionPointContainer **ppCPC);*
*HRESULT Advise(IUnknown *pUnkSink, DWORD *pdwCookie);*
HRESULT Unadvise(DWORD dwCookie);
*HRESULT EnumConnections(IEnumConnections **ppEnum);*

The GetConnectionInterface is used to get the interface connected to this connection point. The GetConnectionPointContainer retrieves the container's implementation of the interface. The container calls Advise function to connect its interface implementation. Unadvise is used to disconnect from the connection point. A connection point may be connected by multiple client sites, each has an implementation of a connected interface. If this is the case, the function EnumConnections can be used to find all of the connections to this connection point.

The container gets the connection point through another interface, IConnectionPointContainer. It has two methods:

*HRESULT EnumConnectionPoints(IEnumConnectionPoints **ppEnum);*
*HRESULT FindConnectionPoint(REFIID iid, IConnectionPoint **ppCP);*

EnumConnectionPoints function can be called to get all of the connection points for a given control. FindConnectionPoint is used to get a particular connection point.

For the ActiveX control's events, the container reads the event type from the control's type library and creates an IDispatch implementation. It then calls the control's connection point's Advise method to hook up its event implementation so that the control can fire events through this implementation.

Data Binding

An ActiveX control, like a regular control, is usually mapped to a data. The control displays the data. If the user changes the control content, its associated data will change too. If the data is changed, the control will reflect the change by displaying the new value. This kind of association is called data binding in ActiveX controls. ActiveX controls' properties can be specified to be bound. When a bound property of an embedded control is changed, its container will decide what action will be taken.

A bound property will send a notification to its container when its value changes. Further, a property can ask its container if a property value can be changed before the property actually being changed. The connection point mechanism is used for property change notification as well. A control supporting data binding will implement a connection point used to hook up the implementation of the IPropertyNotifySink interface. The container will implement this interface and connects it to the control's connection point. The control then notifies all changes to bindable properties through this interface. Furthermore, the same interface is used when a control wants to ask its container if a specific property value can be changed.

The definition of property binding is also in the control's type library. Some attributes are used for data binding. The attribute bindable means that a property will send notifications when it is changed. The attribute requestEdit indicates that a property will call the container's IPropertyNotifySink::OnRequestEdit function before its value is changed. Note that properties that support edit request must also be bindable. The displayBind attribute, which again can only be used when the property is bindable, informs the container that this property should be displayed as a bindable property. The defaultBind attribute tells the container that this property represents the control and therefore should be bound. Only one property per control can have the defaultBind attribute. The edit box would be a typical case for defaultBind.

The interface for data binding is the IPropertyNotifySink which has the following two methods:

HRESULT OnChanged(DISPID dispid);
HRESULT OnRequestEdit(DISPID dispid);

OnChanged function is called by the control after a bound property's value has changed, while OnRequestEdit is used by the control before a property marked with requestEdit is changed. The DISPID is used in these functions to identify the property.

Control and Container Communication

The ActiveX controls specification defines a set of interfaces to allow more specific control and container communication. IOleControl is implemented by the control, while IOleControlSite is implemented by the container.

The IOleControl has four methods:

*HRESULT GetControlInfo(CONTROLINFO *pCI);*
*HRESULT OnMnemonic(MSG *pMsg);*
HRESULT OnAmbientPropertyChange(DISPID dispid);
HRESULT FreezeEvents(BOOL bFreeze);

GetControlInfo is called by the container to get the control's keyboard handling information. OnMnemonic is called by the container when a key in the control's accelerator table is pressed. OnAmbientPropertyChange is called by the container when one or more container's ambient properties is changed. FreezeEvents prevents the control from firing any events.

The IOleControlSite has the following methods:

HRESULT OnControlInfoChanged(void);
HRESULT LockInPlaceActive(BOOL fLock);
*HRESULT GetExtendedControl(IDispatch **ppDis);*
*HRESULT TransformCoords(POINTL *lpptlHimetric,*
*POINTF *lpptfContainer,*
DWORD flags);
*HRESULT TranslateAccelerator(MSG *lpMsg,*
DWORD grfModifiers);
HRESULT OnFocus(BOOL gGotFocus);
HRESULT ShowPropertyFrame(void);

OnControlInfoChanged and TranslateAccelerator functions are client side keyboard handling. LockInPlaceActive is called by a control to let the container to lock the control's in-place active state if such state may cause problem.

GetExtendedControl returns the interface of the extended control implemented by the client site when the control is aggregated. This allows the control to get the current values of its extended properties. OnFocus is called by the control when it is having the focus.

Keyboard Handling

An accelerator is a shortcut keyboard key to initiate a specific event. An accelerator table maintains all the accelerator mapping information for a program. A handle is a variable that identifies an object in the Windows system. A control can tell its container which accelerators it is interested in by passing the container its handle of the accelerator table when the container calls IOleControl::GetControlInfo. If a control dynamically changes this information, it calls IOleControlSite::OnControlInfoChanged which tells the container that it should call IOleControl::GetControlInfo again on that control to update the information.

When a keystroke is received and is recognized as a control's mnemonic, the container calls IOleControl::OnMnemonic to give the control a chance to take an action for the keystroke.

Types and Coordinates

ActiveX control specification defines a set of standard types to represent specific information. For example, OLE_COLOR holds color information by controls and their containers. ActiveX control also defines and implements a standard object for fonts. The standard picture type is used for bitmaps, icons, and metafiles. As for the standard font object, ActiveX control provides an implementation of the standard picture object.

The coordinate system in different containers might be different. Controls may also use coordinates. However, since a control is used inside a container, the container must know which control event parameters are coordinates so that it can convert them into its own

coordinate space. This type of information is also stored in the control's type library by specifying such parameter using one of the standard types defined in OLE.

For coordinate-based properties, the control can ask the container to do the conversion by calling the IOleControlSite::TransformCoords.

Persistence

The control generally needs to remember its state when it is used inside the container so that the state can be restored next time the container application is run and the control is used again. Saving the state is called persistence. When persistence is needed, the container will ask the control to do it through the persistence interface.

Miscellaneous Status Bits

A container might want to know various information about a control before loading and creating the control for efficiency purpose. COM provides this capability through a Registry entry under the CLSID key for the component called MiscStatus..

A container can read the control's MiscStatus bits by calling IOleControl::GetMiscStatus or by reading the Registry entry directly. One such bit is OLEMISC_INSIDEOUT, which tells a container that this control wants to be activated inside-out, i.e., it is capable of being in-place active as well as UI-active.

Another bit is OLEMISC_ACTIVATEWHENVISIBLE which means that this control wants to be activated whenever it becomes visible even if it is not active.

The bit OLEMISC_INVISIBLEATRUNTIME indicates that this control wants to be invisible at run time.

The bit OLEMISC_SIMPLEFRAME is for some controls that are just a holder for other controls. A group box control is an example. A control marked with this bit indicates that it acts as a control container but delegates most of the work up to the real container.

The bit OLEMISC_SETCLIENTSITEFIRST indicates that an ActiveX control can access the client site before their persistent state has been loaded.

Registration

Like all COM/DCOM objects. ActiveX controls need to be registered in the Registry. Besides the entries registered as for all OLE servers. ActiveX controls have some special entries.

The Insertable and Control keys are empty keys. They do not have values associated with them. Insertable means that the control will appear in standard Insert Object dialog boxes presented by the containers. Controls are often not marked as insertable because in many cases controls are useful only in containers that know the controls. The key Control tells the containers that this is a control so that the containers can fill in the dialog box for displaying only the ActiveX controls. This attribute is usually set for the controls.

The DefaultIcon key is used when the control is displayed as an icon. The container uses this key to find the icon to display.

The key ToolboxBitmap32 indicates the bitmap to display for the control in the container's control toolbox.

Property Pages

A property page is a user interface object implemented by the control to allow the control's properties to be view and changed. The control sees property pages as a set of dialog box templates. These templates contain fields for each of the properties that a control wants to expose through property pages and the code behind the dialogs manages the content.

Property pages are COM/DCOM objects with their own CLSIDs and interfaces. This allows property pages to be shared among controls. The property pages of a control are contained in a single property sheet.

5.2 Tools for Creating ActiveX Controls

5.2.1 Microsoft Tools

To ease the task of creating an ActiveX control, various tools can be employed. These tools are developed by Microsoft for different programming purpose. The choices of tools on creating ActiveX controls that are currently available are the following ones[7]:

- C++ and the ActiveX Template Library(ATL);
- Visual Basic;
- Java;

- C++ and Microsoft Foundation Classes (MFC);

Visual Basic and Java use programming languages other than C++, while ATL and MFC both use C++. MFC is designed to make it as easy as possible for the C++ programmer to create controls and applications, whereas ATL is designed to make controls in C++ that are as small and as fast as they can be.

Since using MFC to create an control is relatively easier, an example ActiveX control from Denning[7] will be created using this tool(Visual C++, version 6.0) in this section and various aspects of the control will be enhanced in the subsequent sections.

MFC is built within the Visual C++ tool. Once the tool is installed, the following components on developing controls will be available:

- The MFC run-time libraries;
- An ActiveX control wizard;
- A test container for verifying the control;

5.2.2 Creating a Skeleton Project

The first step in creating a control using MFC is to use the wizard to generate a skeleton control project.

Select New from the File menu. The New dialog will be displayed. Select the Projects tab and choose the MFC ActiveX ControlWizard. Type in the project name, e.g. First and

specify the directory for the project. Click the OK button to go to the first step wizard page. This page presents the following four options:

- The number of controls in this DLL. default: 1;
- Whether the controls have run-time license. default: No;
- Whether you want source code comments. default: Yes;
- Whether you want the help files generated. default: No;

In this example. leave all the defaults as they are. Click Next button to go to the next page of the wizard. This page has the following options:

- The name of the control to apply properties in this wizard page;
- Whether the control should be activated when it is visible. default: Yes;
- Whether the control should be invisible at run time. default: No;
- Whether it appears in a regular Insert Object dialog in containers. default: No;
- Whether it has an About box. default: Yes;
- Whether the control is capable of acting as a simple frame for other control. default: No;
- Which standard Windows window class should be used as base class for this control. if any. default: None.

Again. keep all the defaults. Click the Finish button and confirm the options that have chosen. The MFC ActiveX ControlWizard creates a set of skeleton files for the control project.

5.2.3 The ControlWizard Created Files

The MFC ActiveX ControlWizard has created a set of files used to build the control. First.

First.dsp is the Visual C++, version 6.0, project file and First.dsw is the workspace file. They are used to build the control.
First.clw is used by the Visual C++ ClassWizard to edit existing class or to add new classes. ReadMe.txt is a text file to briefly describe the files that have been generated. First.rc and resource.h are the project resource files. The control's icon is displayed in the control's default About box and is contained in First.ico. Likewise, the control has a bitmap which is used as the toolbox bitmap when the control is embedded in a container such as Visual Basic. This bitmap is contained in FirstCtl.bmp. First.ncb is the file used for displaying ClassView information. Stdafx.h and stdafx.cpp are the standard MFC files for efficient use of precompiled headers.

First.def is the ActiveX control DLL module definition file. It exports the four standard functions for any COM/DCOM in-proc server, that is, DllCanUnloadNow, DllGetClassObject, DllRegisterServer and DllUnregisterServer.

The file First.odl is the automatically generated ODL source file for the control in the project. It defines the type library FIRSTLib as well as the control's dispatch interface, _DFirst. It also defines the control's primary events interface called _DFirstEvents. Finally, the coclass First itself is defined. When compiled, the generated type library is

included in the project's resources. Storing the type library in the control's resources is the recommended way of holding type information.

The remaining files generated in the projects, First.h, First.cpp, FirstCtl.h, FirstCtl.cpp, FirstPpg.h, FirstPpg.cpp are the C++ source files used to define the control. Let's look at them in detail in order to understand the working mechanism behind these files.

5.2.4 The Control Module Class

Appendix 11 lists the files created by the MFC ControlWizard. First.h and First.cpp contain the class CFirstApp. Any MFC user DLL has a CWinApp derived application object. With MFC based ActiveX controls, the application class is actually derived from COleControlModule, which is itself derived from CWinApp.

In First.h, the wizard declares InitInstance and ExitInstance functions. It then declares three global variables as external references because although they are defined in First.cpp, they are used elsewhere and this header file is included elsewhere. The variable _tlid is the GUID of the control's type library, and _wVerMajor and _wVerMinor are control's major and minor version number respectively.

In First.cpp, a global instance of the CFirstApp class is created. Then the tree global variables are initialized. The type library is assigned a GUID and the version is initialize to 1.0.

The InitInstance and ExitInstance functions just call the base class's functions to initialize and uninitialize the application.

DllRegisterServer and DllUnregisterServer are the control DLL's exported function used to register and unregister the DLL as an in-proc server.

In DllRegisterServer and DllUnregisterServer, the macro AFX_MANAGE_STATE is called to set up the module state. A module refers to an executable program, or to a DLL (or set of DLLs) that operate independently of the rest of the application, but uses a shared copy of the MFC DLL. An ActiveX control is a typical example of a module. MFC has state data for each module used in an application. Each module is responsible for correctly switching between module states at all of its entry points such as an exported function or a member function of COM/DCOM interfaces. The actual registration and unregistration are done by calling the appropriate well-defined MFC functions.

5.2.5 The Control Class

The majority of the control's functionality is implemented in the COleControl derived class. In this example, it is CFirstCtrl which is contained in FirstCtl.h and FirstCtl.cpp.

In FirstCtl.h, the class CFirstCtrl uses the macro DECLARE_DYNCREATE to set up the class for dynamic creation. This is the normal MFC standards. This macro is matched by a call to IMPLEMENT_DYNCREATE in the implementation file. Besides the

constructor and the destructor. the class has the drawing function(OnDraw). the property persistence function(DoPropExchange) and the function OnResetState which is called when the control is asked by the container to reset its property values to default ones.

The macro DECLARE_OLECREATE_EX declares the functions to set up the object's class factory. DECLARE_OLETYPELIB declares functions to get an ITypeLib pointer for this control's type library and to implement type library caching. an optimization performed by MFC. DECLARE_PROPPAGEIDS declares a function to retrieve the control's property page CLSIDs. DECLARE_OLECTLTYPE declares functions to get the control's ProgID and miscellaneous status bit values.

The class also declares a few of MFC maps including the message map. the dispatch map and the event map.

Windows applications are basically message-driven programs. A user action will be represented by a message sent to the operating system. and the operating system will handle the message. Due to the large number of messages involved. providing a separate virtual function for each Windows message would result in a huge vtable. MFC provides an alternative to handle messages sent to a window. A mapping from messages to member-functions may be defined so that when a message is to be handled by a window. the appropriate member function is called automatically. This message-map facility is designed to be similar to virtual functions. The DECLARE_MESSAGE_MAP macro declares functions for a class to have a message map table. The message map's table is

defined in the implementation file with a set of macros that expand to message map entries. A table begins with a BEGIN_MESSAGE_MAP macro call, which defines the class that is handled by this message map and the parent class to which unhandled messages are passed. The table ends with the END_MESSAGE_MAP macro call. Between these two macro calls is an entry for each message to be handled by this message map.

MFC establishes a dispatch map for the dispinterface. It designates the internal and external names of object functions and properties, as well as the data types of the properties themselves and of function arguments. DECLARE_DISPATCH_MAP is used in the class declaration to declare that a dispatch map will be used to expose a class's methods and properties. BEGIN_DISPATCH_MAP starts the definition of a dispatch map. END_DISPATCH_MAP ends the definition of a dispatch map. DISP_FUNCTION is used in a dispatch map to define an Automation function. DISP_PROPERTY defines an Automation property.

MFC offers a programming model optimized for firing events. In this model, event maps are used to designate which functions fire which events for a particular control. Event maps contain one macro for each event. DECLARE_EVENT_MAP is used in the class declaration to declare that an event map will be used in a class to map events to event-firing functions. In the implementation file, BEGIN_EVENT_MAP begins the definition of an event map and END_EVENT_MAP ends the definition of an event map.

Finally an enumeration is declared in the header file to hold the DISPIDs for control's properties, methods and events.

In FirstCtl.cpp, the message map only contains one ON_OLEVERB entry for the Properties verb, which will cause the control to display its property pages. The ON_OLEVERB entries allow the control to react to an invocation of one of its verbs, or the OLE actions. Normally, the container places the verb on the Edit menu when an object is selected.

The dispatch map so far only contains the About box entry and the event map is empty. A property page map is also defined starting with BEGIN_PROPPAGEIDS and ending with END_PROPPAGEIDS macros. The numeric parameter in the BEGIN_PROPPAGEIDS macro is the number of pages in the map, and each entry contains the CLSID of the property page.

The IMPLEMENT_OLECREATE_EX macro creates the object's class factory and initializes it with the CLSID and ProgID passed in. Likewise, IMPLEMENT_OLETYPELIB implements the functions for type library with the given GUID and the version number. Then the IIDs of the control's primary dispatch interface and the events interface are defined. Next, the control's miscellaneous bits are initialized. The flag OLEMISC_CANTLINKINSIDE indicates that the control cannot be used as a link source when it is embedded. OLEMISC_RECOMPOSEONRESIZE tells the container that the control likes to re-create its rendition if the size is changed in the

container. IMPLEMENT_OLECTLTYPE macro implements the functions declared in DECLARE_OLECTLTYPE to retrieve the control's ProgID and miscellaneous status bits.

MFC implements the class factory as a nested class, CFirstCtrlFactory, in the control class, CFirstCtrl. The function CFirstCtrl::CFirstCtrlFactory::UpdateRegistry function is implemented by MFC to be used by code elsewhere to register or unregister the control in the system Registry.

The constructor of the control class by now only initialize the IIDs of the dispatch and events interfaces. The OnDraw function is called whenever the control needs to be redrawn. It just draws an ellipse now. Note the parameter to this function. It is the rectangle in which the control can draw something. This rectangle is an area inside the container window. The DoPropExchange is used to transfer persistent property values between the property member variables and the container-provided storage. At present, only the control's version is saved. The base class function call will cause any standard properties used by this control that need saved to actually get saved.

5.2.6 The Property Page Class

The MFC ActiveX ControlWizard creates one blank property page and the property page is wrapped in a class derived from COlePropertyPage. In this example, the page class is called CFirstPropPage which is contained in FirstPpg.h and FirstPpg.cpp files. The class is dynamically creatable and has a class factory since it is a COM object itself. However,

it does not have a type library since it is not an Automation object. It has a function DoDataExchange which is the standard MFC dialog data exchange and validation routine for exchanging data between class member variables and dialog controls. The DDP_PostProcessing call will finish the transfer of property values from the property page to the control when property values are being saved.

5.2.7 Test the Control

The first step is to build the control. Select Rebuild All from the Visual C++ Build menu while the First project is the current project. This will build the control and register it in the system Registry. By convention, an ActiveX control file has the extension of ocx. Hence, the example control is contained in First.ocx.

Test Container(TSTCON32.EXE) is provided in Visual C++ and can be used for quick testing on an ActiveX control. Run the Test Container from Visual C++ Tools menu. After the program is running, choose Insert New Control command from the Edit menu and select First Control from the dialog box. While the control is selected in the container's window, test the following aspects of the control:

Invoke its methods. Test Container learned about the control's methods from its type library. Select Invoke Methods command from the Test Container's Control menu. The only available method is the AboutBox method which will display the control's About box.

Display the control's properties by showing its property pages. Choose Properties ... First Control Object from the Test Container's Edit menu. Currently, only the control's name is displayed in the Extended page and the General page is empty.

If the control provides any events, they can be viewed from the event log by selecting Logging command from the Control menu.

Move and resize the control on the container's window.

Other ActiveX control container programs can also be used to test the control. For example, Visual Basic, Internet Explorer.

5.3 Properties

There are three types of properties for an ActiveX control. Ambient properties are exposed by the client site which the control uses as its environment values. Extended properties are those properties that the container implements on behalf of the control and the control usually does not need to know. Control properties are the ones implemented by the control.

5.3.1 Ambient Properties

ActiveX controls specification predefines a set of standard ambient properties. Some of these are listed below in terms of their names, their symbols and DISPIDs[7]:

BackColor	*DISPID_AMBIENT_BACKCOLOR*	*-701*

DisplayName	*DISPID_AMBIENT_DISPLAYNAME*	*-702*
Font	*DISPID_AMBIENT_FONT*	*-703*
ForeColor	*DISPID_AMBIENT_FORECOLOR*	*-704*
LocaleID	*DISPID_AMBIENT_LOCALEID*	*-705*
MessageReflect	*DISPID_AMBIENT_MESSAGEREFLECT*	*-706*
ScaleUnits	*DISPID_AMBIENT_SCALEUNITS*	*-707*
TextAlign	*DISPID_AMBIENT_TEXTALIGN*	*-708*
SupportsMnemonics	*DISPID_AMBIENT__SUPPORTSMNEMONICS*	*-714*
AutoClip	*DISPID_AMBIENT_AUTOCLIP*	*-715*
Palette	*DISPID_AMBIENT_PALETTE*	*-726*

LocaleID indicates the locale. MessageReflect tells the control that if it is true, the container will reflect the Windows messages back to the control. TextAlign tells the control how it should arrange the text it displays. SupportsMnemonics means that, if it is true, the control can provide mnemonics keypresses and the container will handle them. AutoClip indicates the container will automatically clip the control. Palette returns the handle of the container's color palette. If a container supplies a palette, only this palette can be realized into the foreground.

A container can add more ambient properties and the controls that will be embedded in the container must know these extra ambient properties in order to use them. The control can choose those ambient properties to use that make sense to the control and ignore the others.

5.3.2 Stock Properties

Some control properties are likely to be implemented so often that MFC and other tools provide standard implementation for them. These standard properties are called stock

properties. Again, the implementation of a control can decide to use some of these stock properties and not to use the others.

Any other properties are custom properties that are specific to the control.

5.3.3 Adding Stock Properties

Stock properties can be added in the control by using the MFC tool. For the example control created by MFC tool before, Open the project, First. Invoke ClassWizard and select Automation tab, check that CFirstCtrl class is selected, and click Add Property button. Select BackColor from the External Name combo box and click OK. Repeat for Caption, Enabled, Font, FontColor and hWnd stock properties.

The tool will add description in the properties part of the control's dispinterface in the ODL file, First.odl. For example,

[id(DISPID_BACKCOLOR), bindable, requestedit] OLE_COLOR BackColor;

Note that the property is defined as bindable and requestedit.

The tool will also add entries in the control class' dispatch map, e.g.,

DISP_STOCKPROP_BACKCOLOR()

5.3.4 Adding Custom Properties

A HRESULT is a 32-bit value used as the return value of most of the COM/DCOM interface functions. The highest bit is the severity bit. the lower 16 bits is the error code and the rest of 15 bits is the facility bits.

A control is not much useful unless some custom properties are added to the control. Suppose the example control[7]. First. will be used to convert an existing HRESULT to its component parts. the following custom properties are added: Code having the short type which is the error code of a HRESULT. ErrorName with BSTR type which is the name of the error. Facility with BSTR type which is the facility portion of the HRESULT. Message with BSTR type which is a message associated with the HRESULT. HResult with type long which is the HRESULT itself. Severity with type BSTR which is the severity portion of the HRESULT. Among these. except the HRESULT itself. all other properties are read-only.

Similar to adding stock properties. the custom properties can be added using the ClassWizard. Each property will use the Get/Set Methods implementation scheme. which can be specified in the Add Property dialog box. A read-only property is made by not defining a Set function. which can be achieved by leaving the Set Function edit box blank.

Although the MFC ClassWizard actually allows to add a property as a member variable. or a member variable with a notification. or Get and Set functions. COM/DCOM actually

allows only the Get/Set function implementation scheme. For the other two options. MFC will implement the code to simulate the options.

For each property. the ClassWizard adds a property definition entry in the ODL file. It also adds a dispatch map function declaration and the dispid in the control header file. FirstCtl.h. The dispatch map entry and the function definitions are added in the control implementation file. FirstCtl.cpp. Certainly. these functions need to be modified to perform their tasks.

To test the control. run Visual Basic and add the First control to the project by using Components option on the Project menu. Draw a copy of the First control on the form by selecting the First control bitmap on the toolbox. While the control is selected. look at the Properties window. Those added properties are displayed.

5.4 Property Persistence

A control's properties can be saved to make it persistent. The control has to make a decision on what custom properties should be made persistent. The stock properties will be saved automatically. Also there is little point to save dynamic properties that are recalculated each time they are retrieved.

MFC wraps almost all ActiveX control functionality in the COleControl class. It provides the support for property persistence. When the container asks a control to provide its persistent properties, ultimately the control's DoPropExchange member function is called

to serialize the control's properties. The default override function CFirstCtrl::DoPropExchange generated by the MFC ActiveX ControlWizard will call ExchangeVersion function to persist the version number and call the COleControl::DoPropExchange to persist the stock properties.

The DoPropExchange function is used both to write the properties from the control to the requester and to read them from a previously saved set.

To serialize the custom properties, the DoPropExchange must be modified. MFC provides a set of helper functions to do this task. The names of these functions all begin with PX_. Therefore, they are generally known as PX_ functions. For example, add the following line of code in CFirstCtrl::DoPropExchange function, the HResult property will be made persistent:

PX_Long(pPx, _T("HResult"), m_HResult);

This function takes the MFC provided serialization context object, the name of the property and the value of the property.

In addition to the PX_ functions to handle the obvious types, the PX_Blob function adds persistence support for properties that are BLOBs. A BLOB is a chunk of binary data.

5.5 Methods

A method is a way to ask an object to perform some action. ActiveX controls allow custom methods to be added. A method can take any of the standard types supported by

Automation as arguments and return values of any of these types. In other words, any data types that can be placed in a VARIANT can be used in the ActiveX controls' methods.

Methods can be added to a control by using MFC ClassWizard. Similarly, MFC provides two stock methods. DoClick and Refresh takes no parameters and returns no values. DoClick simulates the action of clicking the left mouse button and Refresh causes the control to be repainted.

For example, to add two methods, a stock method Refresh and a custom method AddHResult, to the example control[7]. First, load the project into Visual C++, invoke the ClassWizard, and select the Automation tab. Make sure that CFirstCtrl is selected as the class name, and click the Add Method button. Add the stock Refresh method by choosing it from the External Name combo box and clicking the OK button. Then add the AddHResult method, choosing BOOL as its return value and giving it three parameters: hResult as a long, Symbol as a LPCTSTR and Message as a LPCTSTR.

As for adding properties, for each method added, the ClassWizard adds a method definition entry in the ODL file. It also adds a dispatch map function declaration and the dispid in the control header file, FirstCtl.h. The dispatch map entry and the function definitions are added in the control implementation file, FirstCtl.cpp. Again, the method requires implementation to perform its task.

Therefore. adding methods to a control is pretty much identical to adding properties. except that with a property. typically two functions(Get and Set) are implemented. a method implements only one. A method is just an Automation method. The important point. as with properties. is to make sure that the description of the interface in the control's ODL file matches the actual implementation so that the type library built from the ODL file accurately reflects the interface.

5.6 Events

Events are a powerful addition to the ActiveX controls. Standard Automation objects can inform their controllers that something has happened only as the result of a call to one of the object's properties or methods as a synchronous response. ActiveX controls. on the other hand. are able to inform their containers at any time. asynchronously. of any action.

There are four types of events: Request events. Before events. After events and Do events[7].

A control fires a Request event to ask its container to allow the control to do something. The last parameter that a Request event takes is a pointer to a variable with type CancelBoolean that is a standard type introduced in ActiveX control. The value of this parameter is set to FALSE by the control prior to the event being fired. If the container changes the parameter to TRUE. then it tells the control not to perform the action signaled by the event. By convention. a Request event has the name beginning with Request. followed by the name of the action. e.g., RequestUpdate.

A Before event is fired by the control before it takes a specific action. giving the container or its user a chance to prepare before the action is taken. A Before event is not cancelable and should be named starting with Before. e.g.. BeforeClose.

An After event is fired by a control to let the container or its user perform some action after the control's action has occurred. After events don't have any naming convention and they are not cancelable. A typical After event is a mouse click in which a control notifies its container that it has been clicked.

A Do event allows the container or its user to override an action or to perform some behavior immediately before the default action is taken. If a Do event has a default action that it can take. it passes in the event as the last parameter a pointer to a variable of type Boolean. conventionally called EnableDefault. which is set to TRUE by the control before the event is fired. If the container sets this parameter to FALSE. it asks the control not to perform its default action. Do events typically are named starting with Do. followed by the name of the action.

Currently. a control fires an event by calling the Invoke method on the dispinterface pointer it extracted from the container when the container connected to its connection point through IConnectionPoint::Advise.

The ActiveX Controls specification defines a set of standard events that controls can choose to fire. This assures that certain common events have the predefined semantics.

Custom events should use positive dispids so that they do not conflict with standard events, methods, or properties.

The following list shows the names, dispids and description of some standard events[7]:

Click	*-600*	*control is clicked.*
DblClick	*-601*	*control is double clicked.*
KeyDown	*-602*	*a key is pressed while a control has focus.*
KeyUp	*-604*	*a key is released while a control has focus.*
MouseDown	*-605*	*mouse button is down over a control.*
MouseMove	*-606*	*mouse is moved over a control.*
MouseUp	*-607*	*mouse button is released over a control.*
Error	*-608*	*an error has occurred.*
ReadyStateChange	*-609*	*used during loading properties and the data received by a control causes it to transition to the next ready state.*

MFC implements the previous standard events as stock events, any other events are treated as custom events. To add an event, invoke the ClassWizard, click the ActiveX Events tab. Select the class to which to add events and click the Add Event button.

After adding a stock event, e.g., Click, the ClassWizard will add an entry in the method part of the event dispinterface of the control's ODL file as:

[id(DISPID_CLICK)] void Click();

It will also add a macro EVENT_STOCK_CLICK() to the class's event map in the control's implementation file. The implementation function for firing Click event is FireClick which can now be called anywhere within the control code to fire the Click event. The FireClick is a member function of COleControl class which calls, via a helper function, the Invoke method the IDispatch interfaces attached to the control's event connection point. If nothing is connected to the connection point, no event gets fired.

If a custom event is added through the ClassWizard, the ClassWizard adds a function prefixed with Fire to the class's event map in the header file. For example, adding a custom event, InvalidHResult with a long integer as its parameter, would add an entry in the control's event dispinterface in the ODL file as

[id(1)] void InvalidHResult(long HResult);

and a definition entry in the control's event map in the header file as

void FireInvalidHResult(long HResult)
{FireEvent(eventidInvalidHResult,EVENT_PARAM(VTS_I4), HResult);}

Basically, the FireEvent function is used for all event invocations, including stock events. It is a member function of COleControl which takes a variable number of parameters since it is a generic function designed to handle all event invocations.

Also a dispid is generated in the control's header file for this event and another entry is inserted in the event map in the control's implementation file as

EVENT_CUSTOM("InvalidHResult", FireInvalidHResult, VTS_I4)

which defines the event name, the function that fires it and the parameters it takes.

After an event-firing function is defined in the event map, it can be called at any point within the control to cause the event fired up to its container. However, there is no guarantee that a container is always ready to handle the event.

When a container connects to a control's event connection point, it calls IConnectionPoint::Advise. This is sent on to the control's instance of the COleControl class by MFC to a member function called OnEventAdvise. This function will get called

each time a connection is made. It also gets called each time a connection is broken. Another COleControl virtual function, OnFreezeEvent, is called whenever a control's IOleControl::FreezeEvents method is called.

5.7 Property Pages

Property pages as defined by the current user interface paradigm are sets of related dialogs that appear as a number of pages in a tabbed dialog box. However, each page is a COM object.

Every control created with the MFC ActiveX ControlWizard has a property page unless it is deliberately removed. The property page is the user interface for the control to present their properties for access. You can add controls to the page, add new pages, and tie the controls on each page to specific properties exposed by the control. MFC's property page functionality is wrapped in a class called COlePropertyPage, from which the MFC ActiveX ControlWizard and ClassWizard derive specific classes to wrap a given control's property pages.

The first step in adding a field to a property page is deciding what type of control to use to present the property. For example, for the HResult property in the example control[7], the most logical representation is an edit box, but the content must be numeric as the property has the type long. The following steps add the controls for HResult property in the control's property page:

- Using Visual C++ Resource editor, edit the dialog resource called IDD_PROPPAGE_FIRST by adding to it a static text with &HResult: as the text. The & character signifies that the character following it is the mnemonic character in the dialog box.
- Add an edit box beside the static text.
- Invoke the ClassWizard, select the Member Variables tab, and ensure that CFirstPropPage class is selected, select the control id, IDC_EDIT1, which is the id for the newly added edit box. Click Add Variable button.
- Enter a name for the variable, e.g., m_HResult. Make sure that the Category box says Value. Choose long as the variable's type.
- Type in HResult as the OLE Property Name. Click OK button to finish.

The ClassWizard adds in the header file of the property page class, FirstPpg.h, a member variable like

long m_HResult;

This member variable will be initialized in the page class's constructor. The binding between the edit box and the member variable as well as the ActiveX control's property is done in the page's DoDataExchange function, the MFC standard dialog data exchange and validation routine. Two lines of code is inserted in this function's data map:

```
DDP_Text(pDX, IDC_EDIT1, m_HResult, _T("HResult") );
DDX_Text(pDX, IDC_EDIT1, m_HResult);
```

Extra property pages can also be added if there are too many controls to fit on one page. To add a new page, a new dialog resource and a new class to wrap the dialog must be created. Then increment the page count in the BEGIN_PROPPAGEIDS macro located in

the control's implementation file. e.g., FirstCtl.cpp. Add an entry in between the BEGIN_PROPPAGEIDS and END_PROPPAGEIDS macros for the new page. Implement the new page class as appropriate. The new page will be displayed when the ActiveX control's property is displayed.

Some common property pages, known as stock property pages, are already implemented and provided by MFC. These are for colors, fonts and pictures. The stock pages can be added by modifying the code block in the control's implementation file between the BEGIN_PROPPAGEIDS and END_PROPPAGEIDS macros. Add the following entries to add the stock pages:

PROPPAGEID(CLSID_CColorPropPage)
PROPPAGEID(CLSID_CFontPropPage)
PROPPAGEID(CLSID_CPicturePropPage)

The stock pages will interrogate the control's type library to learn which properties have the requisite types and display them on the appropriate page.

6 Conclusions

The component-based programming technologies surveyed in this report, namely, DLL, COM/DCOM, Automation and ActiveX controls, are important technologies for software reuse. More and more applications are developed utilizing these technologies. They provide a consist approach for creating different small binary software objects. These components can be used by the end user as building blocks to create different applications at run time.

However, creating components without using tools is cumbersome, and the tools for creating components cannot be easily mastered. As C++ is a natural language for writing object-oriented software applications, a programming language suited for creating components is certainly more desired.

References:

1. "The Advantages of Using DLLs", Visual C++ Programmer's Guide, MSDN Library, April, 1999.
2. "The Component Object Model Specification ", Draft Version 0.9, October 24, 1995. Microsoft Corporation and Digital Equipment Corporation.
3. "Inside COM", Dale Rogerson, Microsoft Press,1997.
4. "Three Pointer Types", MSDN Library, October 1996.
5. "Inside OLE", Kraig Brockschmidt, Microsoft Press, 1995
6. "Professional DCOM Programming", Dr. Richard Grimes, Wrox Press Ltd. 1997.
7. "ActiveX Controls Inside Out", Adam Denning, Microsoft Press, 1997.
8. "Inside Visual C++", David J. Kruglinski, Microsoft Press, 1996.
9. "Teach Yourself Visual Basic 4 in 21 Days", Nathan Gurewich and Ori Gurewich, Sams Publishing, 1995.

Appendix 1 A DLL Example

```
==============================================
/*dllExample.h
 Declarations of the DLL functions. It will be used
 by both of the implementation file of the DLL and
 the client application of this DLL. */

/* To prevent multiple inclusion */
#ifndef DLLEXAMPLE_H
#define DLLEXAMPLE_H

/* The extern "C" block is defined if C++ compiler is used.
  It will turn off the C++ name mangling to the exported API
  names and make them usable by non-C++ client applications. */

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#ifdef DLLEXAMPLE_C

    /* The implementation file for this declaration will
     define DLLEXAMPLE_C, hence the function is declared with
     __declspec(dllexport) modifier and make it an exported API. */

   __declspec(dllexport) void dllExampleFunction(void);
   __declspec(dllexport) int  dllParamFunction(int n, char *str,
                        float data[3], float *sum);

#else

    /* The client application will not define DLLEXAMPLE_C.
     When the client include this header file, the
     function is declared with __declspec(dllimport)
     modifier and make it an imported API from the DLL. */

   __declspec(dllimport) void dllExampleFunction(void);
   __declspec(dllimport) int  dllParamFunction(int n, char *str,
                        float data[3], float *sum);

#endif /* DLLEXAMPLE_C */

#ifdef __cpluplus
}
#endif /* __cplusplus */

#endif /* DLLEXAMPLE_H */
```

```
================================================
/* dllExample.c.
  Implementation of the DLL function. It will
  define DLLEXAMPLE_C so that the right function prototype
  is declared when dllExample.h is included. */

#define DLLEXAMPLE_C

#include "dllExample.h"
#include "windows.h"
#include "stdio.h"

void dllExampleFunction(void)
{
   MessageBox(NULL. "The example DLL API is called!".
          "Message". MB_SYSTEMMODAL):
}

int dllParamFunction(int n. char *str.
              float data[3]. float *sum)
{
  printf("interger parameter: %d\n". n):
  printf("string parameter: %s\n". str):
  printf("data 1= %g\ndata 2= %g\ndata 3= %g\n".
                   data[0]. data[1]. data[2]):
         *sum= data[0] + data[1] + data[2]:
         return 1:
}
```

Appendix 2 An Client Example Using a DLL Statically

```
=================================================
 * dllClientExample.c
  Implementation of an example client application to use
  DLLExample.dll. It will statically link to the dll's
  import library DLLExample.lib when the application stars up. *

#include "dllExample.h"
#include "windows.h"

void main()
{
  int n= 12345;
  char *str= "Example String Parameter";
  float data[3]= { 1.1, 2.2, 3.3 };
  float sum;
  int ret;

  MessageBox(NULL, "Calling a DLL function...", "Message", MB_SYSTEMMODAL);
  dllExampleFunction();

        MessageBox(NULL, "Calling a DLL function with parameters...",
          "Message", MB_SYSTEMMODAL);

  ret= dllParamFunction(n, str, data, &sum);
        if ( ret == 1 )
        {
          printf("\nThe sum is: %g", sum);
    MessageBox(NULL, "Function called successfully...",
           "Message", MB_SYSTEMMODAL);
        }
        else
          MessageBox(NULL, "Function call failed...",
           "Message", MB_SYSTEMMODAL);
}
```

Appendix 3 A Client Example Using a DLL Dynamically

```
==================================================
 * dllClientDynamic.c
  Implementation of an example client application to use
  DLLExample.dll. It will dynamically load the dll, use the
  DLL's API and free the DLL. *

#include "windows.h"

typedef void (EXAMPLEPROC)(void);
void main()
{
   HINSTANCE hModule= NULL;
   EXAMPLEPROC* pFunc= NULL;

   MessageBox(NULL, "Loading the example DLL...",
         "Message", MB_SYSTEMMODAL);

   hModule= LoadLibrary("DLLExample.dll");
   if (hModule != NULL)
   {
      MessageBox(NULL, "Getting the function address...",
         "Message", MB_SYSTEMMODAL);
      pFunc= (EXAMPLEPROC*) GetProcAddress((HMODULE)hModule,
                  "dllExampleFunction");
      if (pFunc != NULL)
         (*pFunc)();

      MessageBox(NULL, "Releasing the example DLL...",
         "Message", MB_SYSTEMMODAL);
      FreeLibrary(hModule);
   }
}
```

Appendix 4 A Visual Basic Example Using a DLL

```
===============================================
VERSION 5.00
Begin VB.Form frmDLLClient
  Caption        =  "DLL Client"
  ClientHeight   =  3195
  ClientLeft     =  60
  ClientTop      =  345
  ClientWidth    =  4680
  LinkTopic      =  "Form1"
  ScaleHeight    =  3195
  ScaleWidth     =  4680
  StartUpPosition =  3 'Windows Default
  Begin VB.CommandButton.cmdUseDLL
    Caption      =  "Use DLL..."
    Height       =  495
    Left         =  1560
    TabIndex     =  1
    Top          =  840
    Width        =  1215
  End
  Begin VB.CommandButton cmdExit
    Caption      =  "E&xit"
    Height       =  495
    Left         =  1560
    TabIndex     =  0
    Top          =  1920
    Width        =  1215
  End
End
Attribute VB_Name = "frmDLLClient"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
'Call a DLL procedure.
'See VB Help for more information on Declare statement.

Private Declare Sub dllExampleFunction Lib "DLLExample.dll" ()


Private Sub cmdExit_Click()
  End
End Sub

Private Sub cmdUseDLL_Click()
  Call dllExampleFunction
End Sub
```

Appendix 5 An Interface Example

```
==============================================
.*
 An example of two interfaces defined using two pure
 abstract base classes in C++.

 The word interface is defined as struct.

 A pure abstract base class which only contains a set of
 pure virtual functions can be used to define an interface.

 The __stdcall specifies that the function uses the standard.
 also known as the Pascal. calling convention where the called
 function clean up the stack frame. Virtually all functions
 offered by COM DCOM interfaces on Microsoft platforms use the
 standard calling convention.

 A component written in C++ class can inherit the pure abstract
 base classes to implement the interfaces that the component
 wants to support.

*

#include "objbase.h" . where 'interface' is defined

interface IInterfaceX
{
  virtual void __stdcall FuncX1(void) = 0;
  virtual void __stdcall FuncX2(void) = 0;
};

interface IInterfaceY
{
  virtual void __stdcall FuncY1(void) = 0;
  virtual void __stdcall FuncY2(void) = 0;
};

class CExampleComponent : public IInterfaceX.
                  public IInterfaceY
{
public:
  //implementation of interface IInterfaceX
  virtual void __stdcall FuncX1(void)
    { cout << "Implementation of FuncX1" << endl; }
  virtual void __stdcall FuncX2(void)
    { cout << "Implementation of FuncX2" << endl; }

  //implementation of interface IInterfaceY
  virtual void __stdcall FuncY1(void)
    { cout << "Implementation of FuncY1" << endl; }
  virtual void __stdcall FuncY2(void)
    { cout << "Implementation of FuncY2" << endl; }
};
```

Appendix 6 An Example to Query Interfaces

```
=================================================
.* QueryInterfaceExample.cpp
  Implementation of standard QueryInterface function
  and the usage of QueryInterface in an example component *

#include "iostream.h"
#include "objbase.h"
#include "windows.h"

.' helper function used in this file
void displayMessage(const char *msg) { cout << msg << endl: }

  Interfaces IX. IY and IZ all inherit IUnknown
  interface to conform to the COM/DCOM specification
interface IX : IUnknown
{
  virtual void __stdcall Fx(void) = 0:
}:

interface IY : IUnknown
{
  virtual void __stdcall Fy(void) = 0:
}:

interface IZ : IUnknown
{
  virtual void __stdcall Fz(void) = 0:
}:

.. Forward declarations for interface ids
extern const IID IID_IX:
extern const IID IID_IY:
extern const IID IID_IZ:

  An example component that supports IX and IY
  interfaces. It will implement all function that
'. declared in the interfaces it supports.
class CA : public IX.
       public IY
{
  /. IUnknown interface implementation
  virtual HRESULT __stdcall QueryInterface(const IID &iid.
                              void **ppv):
  virtual ULONG __stdcall AddRef()  { return 0: }
  virtual ULONG __stdcall Release() { return 0: }

  // IX interface implementation
  virtual void __stdcall Fx(void) { displayMessage("Fx function is called!"): }

    IY interface implementation
  virtual void __stdcall Fy(void) { displayMessage("Fy function is called!"): }
}:
```

```
// Implementation of QueryInterface for the example component
HRESULT __stdcall CA::QueryInterface
  (const IID &iid,  //which interface to get?
   void **ppv)      //the returned interface or NULL
{
  if (iid == IID_IUnknown)
  {
    displayMessage("QueryInterface returns pointer to IUnknown!");
    *ppv= static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    displayMessage("QueryInterface returns pointer to IX!");
    *ppv= static_cast<IX*>(this);
  }
  else if (iid == IID_IY)
  {
    displayMessage("QueryInterface returns pointer to IY!");
    *ppv= static_cast<IY*>(this);
  }
  else
  {
    displayMessage("This component does not support the interface!");
    *ppv= NULL;
    return E_NOINTERFACE;
  }

  static_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}


// Creation function
IUnknown *CreateInstance()
{
  IUnknown *pI= static_cast<IX*>(new CA);
  pI->AddRef();
  return pI;
}

//
// IIDs
//
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IX =
        {0x32bb8320, 0xb41b, 0x11cf,
        {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IY =
        {0x32bb8321, 0xb41b, 0x11cf,
        {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
```

```
static const IID IID_IZ =
        {0x32bb8322, 0xb41b, 0x11cf,
        {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

  An example for the client of the component
int main()
{
  HRESULT hr;

   creates and get the returned IUnknown pointer
  displayMessage("Client creates an instance of the component!");
  IUnknown *pIUnknown= CreateInstance();

   get IX interface from IUnknown interface
  IX *pIX= NULL;
  hr= pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
  if (SUCCEEDED(hr))
  {
    displayMessage("Succeed to get IX from IUnknown!");
    pIX->Fx();   use interface IX
  }

   get IY interface from IUnknown interface
  IY *pIY= NULL;
  hr= pIUnknown->QueryInterface(IID_IY, (void**)&pIY);
  if (SUCCEEDED(hr))
  {
    displayMessage("Succeed to get IY from IUnknown!");
    pIY->Fy();   use interface IY
  }

   get an unsupported interface
  IZ *pIZ= NULL;
  hr= pIUnknown->QueryInterface(IID_IZ, (void**)&pIZ);
  if (SUCCEEDED(hr))
  {
    displayMessage("Succeed to get IZ from IUnknown!");
    pIZ->Fz();   use interface IZ
  }
  else
    displayMessage("Failed to get IZ from IUnknown!");

   get interface IX from itself
  IX *pIXFromIX= NULL;
  hr= pIX->QueryInterface(IID_IX, (void**)&pIXFromIX);
  if (SUCCEEDED(hr))
  {
    displayMessage("Are the IX interface pointers equal?");
    if (pIXFromIX == pIX)
      displayMessage("Yes, they are equal!");
    else
      displayMessage("No, they are not equal!");

  }

   get interface IY from interface IX
```

```
IY *pIYFromIX= NULL:
hr= pIX->QueryInterface(IID_IY. (void**)&pIYFromIX):
if (SUCCEEDED(hr))
{
   displayMessage("Succeed to get IY from IX!"):
   pIY->Fy(): //use interface IY
}

// get interface IUnknown from interface IY
IUnknown *pIUnknownFromIY= NULL:
hr= pIY->QueryInterface(IID_IUnknown. (void**)&pIUnknownFromIY):
if (SUCCEEDED(hr))
{
   displayMessage("Are the IUnknown interface pointers equal?"):
   if (pIUnknownFromIY == pIUnknown)
      displayMessage("Yes. they are equal!"):
   else
      displayMessage("No. they are not equal!"):

}

// delete the component
delete pIUnknown:

MessageBox(NULL.
   "When finish viewing the execution result. click OK to terminate!".
   "Message". MB_SYSTEMMODAL):

return 0:
}
```

Appendix 7 A Component Lifetime Control Example

```
=================================================
 * ReferenceCountExample.cpp
  Implementation of an example component with
  reference counting *

#include "iostream.h"
#include "objbase.h"
#include "windows.h"

// helper function used in this file
void displayMessage(const char *msg) { cout << msg << endl; }

// Interfaces IX, IY all inherit IUnknown
// interface to conform to the COM specification
interface IX : IUnknown
{
  virtual void __stdcall Fx(void) = 0;
};

interface IY : IUnknown
{
  virtual void __stdcall Fy(void) = 0;
};

// Forward declarations for interface ids
extern const IID IID_IX;
extern const IID IID_IY;

// An example component that supports IX and IY
// interfaces. It will implement all function that
// declared in the interfaces it supports.
class CA : public IX,
       public IY
{
  // IUnknown interface implementation
  virtual HRESULT __stdcall QueryInterface(const IID &iid,
                        void **ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  // IX interface implementation
  virtual void __stdcall Fx(void) { displayMessage("Fx function is called!"); }

  // IY interface implementation
  virtual void __stdcall Fy(void) { displayMessage("Fy function is called!"); }

public:
  CA() : m_cRef(0) { }
  ~CA() { displayMessage("CA component destroys itself"); }

private:
  long m_cRef; //the reference count used for all interfaces
```

```
};

// Implementation of QueryInterface for the example component
HRESULT __stdcall CA::QueryInterface
  (const IID &iid,   // which interface to get?
   void **ppv)       // the returned interface or NULL
{
  if (iid == IID_IUnknown)
  {
    displayMessage("QueryInterface returns pointer to IUnknown!");
    *ppv= static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    displayMessage("QueryInterface returns pointer to IX!");
    *ppv= static_cast<IX*>(this);
  }
  else if (iid == IID_IY)
  {
    displayMessage("QueryInterface returns pointer to IY!");
    *ppv= static_cast<IY*>(this);
  }
  else
  {
    displayMessage("This component does not support the interface!");
    *ppv= NULL;
    return E_NOINTERFACE;
  }

  static_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

// The implementation of AddRef
ULONG __stdcall CA::AddRef()
{
  cout << "CA: AddRef= " << m_cRef+1 << endl;
  return InterlockedIncrement(&m_cRef);
}

// The implementation of Release
ULONG __stdcall CA::Release()
{
  cout << "CA: Release= " << m_cRef-1 << endl;
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}

// Creation function
IUnknown *CreateInstance()
{
```

```
    IUnknown *pI= static_cast<IX*>(new CA);
    pI->AddRef();
    return pI;
 }

 IIDs

 {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IX =
          {0x32bb8320, 0xb41b, 0x11cf,
          {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

 {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IY =
          {0x32bb8321, 0xb41b, 0x11cf,
          {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

 An example for the client of the component
int main()
{
   HRESULT hr;

   creates and get the returned IUnknown pointer
   displayMessage("Client creates an instance of the component!");
   IUnknown *pIUnknown= CreateInstance();

   get IX interface from IUnknown interface
   IX *pIX= NULL;
   hr= pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
   if (SUCCEEDED(hr))
   {
     displayMessage("Succeed to get IX from IUnknown!");
     pIX->Fx(); //use interface IX
     pIX->Release();
   }

   get IY interface from IUnknown interface
   IY *pIY= NULL;
   hr= pIUnknown->QueryInterface(IID_IY, (void**)&pIY);
   if (SUCCEEDED(hr))
   {
     displayMessage("Succeed to get IY from IUnknown!");
     pIY->Fy(); //use interface IY
     pIY->Release();
   }

   displayMessage("Client: Release IUnknown interface");
   pIUnknown->Release();

   MessageBox(NULL,
     "When finish viewing the execution result, click OK to terminate!",
     "Message", MB_SYSTEMMODAL);

   return 0;
}
```

Appendix 8 Use A Component Implemented in a DLL

```
==================================================
 ComponentExport.h
 Declaration of the exported function in the DLL
 so that the client of the DLL can use it

#ifndef COMPONENTEXPORT_H
#define COMPONENTEXPORT_H

extern "C"
{

#define DLLENTRY __declspec(dllexport)
DLLENTRY IUnknown* CreateInstance(void);

}

#endif  COMPONENTEXPORT_H


==================================================
 IInterface.h
 Declaration of interfaces

interface IX : IUnknown
{
        virtual void __stdcall Fx() = 0 ;
};

interface IY : IUnknown
{
        virtual void __stdcall Fy() = 0 ;
};

interface IZ : IUnknown
{
        virtual void __stdcall Fz() = 0 ;
};

 Forward declarations for GUIDs
extern "C"
{
        extern const IID IID_IX ;
        extern const IID IID_IY ;
        extern const IID IID_IZ ;
}


==================================================
 GUIDs.cpp
 Interface IDs

#include "objbase.h"
```

```
extern "C"
{
	// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
	extern const IID IID_IX =
		{0x32bb8320, 0xb41b, 0x11cf,
		{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

	// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
	extern const IID IID_IY =
		{0x32bb8321, 0xb41b, 0x11cf,
		{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

	// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
	extern const IID IID_IZ =
		{0x32bb8322, 0xb41b, 0x11cf,
		{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

	// The extern is required to allocate memory for C++ constants.
}


//================================================
// Component.cpp
// Implementation file of an example component packaged
// in a DLL as an in-proc COM server.

#define COMPONENT_CPP

#include "iostream.h"
#include "objbase.h"

#include "IInterface.h"
#include "ComponentExport.h"

void trace(const char* msg) { cout << "Component CA:\t" << msg << endl ;}

// Example Component that supports interface IX and IY
class CA : public IX,
	public IY
{
	// IUnknown implementation
	virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
	virtual ULONG __stdcall AddRef();
	virtual ULONG __stdcall Release();

	// Interface IX implementation
  virtual void __stdcall Fx() { trace("Fx is called!"); }

	// Interface IY implementation
	virtual void __stdcall Fy() { trace("Fy is called!"); }

public:
	// Constructor
	CA() : m_cRef(0) { }
```

```
      Destructor
     ~CA() { trace("Destroy self."); }

private:
   long m_cRef;
};

HRESULT __stdcall CA::QueryInterface
   (const IID &iid,
    void **ppv)
{
         if (iid == IID_IUnknown)
         {
                   trace("Return pointer to IUnknown.");
                   *ppv = static_cast<IX*>(this);
         }
         else if (iid == IID_IX)
         {
                   trace("Return pointer to IX.");
                   *ppv = static_cast<IX*>(this);
         }
         else if (iid == IID_IY)
         {
                   trace("Return pointer to IY.");
                   *ppv = static_cast<IY*>(this);
         }
         else
         {
                   trace("Interface not supported.");
                   *ppv = NULL;
                   return E_NOINTERFACE;
         }
         reinterpret_cast<IUnknown*>(*ppv)->AddRef();
         return S_OK;
}

ULONG __stdcall CA::AddRef()
{
         return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CA::Release()
{
         if (InterlockedDecrement(&m_cRef) == 0)
         {
                   delete this;
                   return 0;
         }
         return m_cRef;
}

// Creation function
extern "C" DLLENTRY IUnknown* CreateInstance(void)
{
         IUnknown* pI= static_cast<IX*>(new CA);
         pI->AddRef();
```

```
        return pI;
}


//=============================================================
// Create.h
//

#ifndef CREATE_H
#define CREATE_H

IUnknown* CallCreateInstance(char* name);

#endif // CREATE_H


//=============================================================
// Create.cpp
// Example client implementation file for a function to
// load the given DLL, create the component and return
// the IUnknown interface pointer of the component.

#define CREATE_CPP

#include "iostream.h"
#include "unknwn.h"    // Declare IUnknown.

#include "Create.h"

typedef IUnknown* (*CREATEFUNCPTR)();

IUnknown* CallCreateInstance(char* name)
{
   // Load dynamic link library into process.
   cout << "Loading DLL" << name << "..." << endl;
   HINSTANCE hModule= ::LoadLibrary(name) ;
   if (hModule == NULL)
   {
         cout << "Error: Cannot load " << name << endl;
         return NULL ;
   }

   // Get address for CreateInstance function in the DLL
   CREATEFUNCPTR CreateInstance
         = (CREATEFUNCPTR)::GetProcAddress(hModule, "CreateInstance");
   if (CreateInstance == NULL)
   {
         cout  << "CallCreateInstance:\tError: "
             << "Cannot find CreateInstance function."
             << endl ;
         return NULL ;
   }

   return CreateInstance() ;
}
```

===

```
// Client.cpp
#define CLIENT_CPP

#include "iostream.h"
#include "objbase.h"
#include "windows.h"

#include "IInterface.h"
#include "Create.h"

void trace(const char* msg) { cout << "Client :\t" << msg << endl; }

int main()
{
	HRESULT hr ;

	// Create component by calling the CreateInstance function in the DLL.
	trace("Get an IUnknown pointer.");
	IUnknown* pIUnknown = CallCreateInstance("ComponentDLL.dll");
	if (pIUnknown == NULL)
	{
		trace("CallCreateInstance Failed.");
		return 1;
	}

	trace("Get interface IX.");
	IX* pIX;
	hr= pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
	if (SUCCEEDED(hr))
	{
		trace("Succeeded getting IX.");
		pIX->Fx(); // Use interface IX.
		pIX->Release();
	}
	else
		trace("Could not get interface IX.");

	trace("Get interface IY.");

	IY* pIY;
	hr= pIUnknown->QueryInterface(IID_IY, (void**)&pIY);
	if (SUCCEEDED(hr))
	{
		trace("Succeeded getting IY.");
		pIY->Fy(); // Use interface IY.
		pIY->Release();
	}
	else
		trace("Could not get interface IY.");

	trace("Release IUnknown interface.");
	pIUnknown->Release();

 MessageBox(NULL,
```

```
    "When finish viewing the execution result. click OK to terminate!".
    "Message". MB_SYSTEMMODAL):

        return 0;
}
```

Appendix 9 An In-proc Server Component Example

```
=================================================
// Iface.h
// Declarations of interfaces, IIDs, and CLSID
// shared by the client and the component.

#ifndef IFACE_H
#define IFACE_H

interface IX : IUnknown
{
  virtual HRESULT __stdcall Fx() = 0;
};

interface IY : IUnknown
{
  virtual HRESULT __stdcall Fy() = 0 ;
};


interface IZ : IUnknown
{
  virtual HRESULT __stdcall Fz() = 0 ;
};

// Declaration of GUIDs for interfaces and component.
// These constants are defined in GUIDs.cpp.

extern "C" const IID IID_IX ;
extern "C" const IID IID_IY ;
extern "C" const IID IID_IZ ;

extern "C" const CLSID CLSID_Component1 ;

#endif // IFACE_H


=================================================
// GUIDs.cpp
// Defines all IIDs and CLSIDs for the client and the component.
// The declaration of these GUIDs is in Iface.h

#define GUIDS_CPP

#include "objbase.h"

// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
extern "C" const IID IID_IX =
        {0x32bb8320, 0xb41b, 0x11cf,
        {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
extern "C" const IID IID_IY =
```

```
        {0x32bb8321. 0xb41b. 0x11cf.
        {0xa6. 0xbb. 0x0. 0x80. 0xc7. 0xb2. 0xd6. 0x82}} :

 {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
extern "C" const IID IID_IZ =
        {0x32bb8322. 0xb41b. 0x11cf.
        {0xa6. 0xbb. 0x0. 0x80. 0xc7. 0xb2. 0xd6. 0x82}} :

 {0c092c21-882c-11cf-a6bb-0080c7b2d682}
extern "C" const CLSID CLSID_Component1 =
        {0x0c092c21. 0x882c. 0x11cf.
        {0xa6. 0xbb. 0x0. 0x80. 0xc7. 0xb2. 0xd6. 0x82}} :


==============================================
 Registry.h
 Helper functions registering and unregistering a component.

#ifndef REGISTRY_H
#define REGISTRY_H

 This function will register a component in the Registry.
 The component calls this function from its DllRegisterServer function.
HRESULT RegisterServer(HMODULE hModule.
                const CLSID &clsid.
                const char *szFriendlyName.
                const char *szVerIndProgID.
                const char *szProgID) :

 This function will unregister a component. Components
 call this function from their DllUnregisterServer function.
HRESULT UnregisterServer(const CLSID &clsid.
                const char *szVerIndProgID.
                const char *szProgID) :

#endif  REGISTRY_H


==============================================
 Registry.cpp

#include "objbase.h"
#include "assert.h"
#include "iostream.h"

#include "Registry.h"

//////////////////////////////////////////
// Internal helper functions prototypes

// Set the given key and its value.
BOOL setKeyAndValue(const char* pszPath.
            const char* szSubkey.
            const char* szValue):

// Convert a CLSID into a char string.
```

```
void CLSIDtochar(const CLSID& clsid,
            char* szCLSID,
            int length);

// Delete szKeyChild and all of its descendents.
LONG recursiveDeleteKey(HKEY hKeyParent, const char* szKeyChild);


// Size of a CLSID as a string
const int CLSID_STRING_SIZE = 39;

///////////////////////////////////////////////////////////
// Public function implementation

// Register the component in the registry.
HRESULT RegisterServer(HMODULE hModule,            // DLL module handle
               const CLSID& clsid,         // Class ID
               const char* szFriendlyName, // Friendly Name
               const char* szVerIndProgID, // Programmatic
               const char* szProgID)       //   IDs
{
        // Get server location.
        char szModule[512];
        DWORD dwResult =
                ::GetModuleFileName(hModule,
                                szModule,
                                sizeof(szModule)/sizeof(char));
        assert(dwResult != 0);

        // Convert the CLSID into a char.
        char szCLSID[CLSID_STRING_SIZE];
        CLSIDtochar(clsid, szCLSID, sizeof(szCLSID));

        // Build the key CLSID\\{...}
        char szKey[64];
        strcpy(szKey, "CLSID\\");
        strcat(szKey, szCLSID);

        // Add the CLSID to the registry.
        setKeyAndValue(szKey, NULL, szFriendlyName);

        // Add the server filename subkey under the CLSID key.
        setKeyAndValue(szKey, "InprocServer32", szModule);

        // Add the ProgID subkey under the CLSID key.
        setKeyAndValue(szKey, "ProgID", szProgID);

        // Add the version-independent ProgID subkey under CLSID key.
        setKeyAndValue(szKey, "VersionIndependentProgID",
                szVerIndProgID);

        // Add the version-independent ProgID subkey under HKEY_CLASSES_ROOT.
        setKeyAndValue(szVerIndProgID, NULL, szFriendlyName);
        setKeyAndValue(szVerIndProgID, "CLSID", szCLSID);
        setKeyAndValue(szVerIndProgID, "CurVer", szProgID);
```

```
	// Add the versioned ProgID subkey under HKEY_CLASSES_ROOT.
	setKeyAndValue(szProgID, NULL, szFriendlyName);
	setKeyAndValue(szProgID, "CLSID", szCLSID);

	return S_OK;
}


// Remove the component from the registry.
LONG UnregisterServer(const CLSID& clsid,         // Class ID
                      const char* szVerIndProgID, // Programmatic
                      const char* szProgID)       //   IDs
{
	// Convert the CLSID into a char.
	char szCLSID[CLSID_STRING_SIZE];
	CLSIDtochar(clsid, szCLSID, sizeof(szCLSID));

	// Build the key CLSID\\{...}
	char szKey[64];
	strcpy(szKey, "CLSID\\");
	strcat(szKey, szCLSID);

	// Delete the CLSID Key - CLSID\{...}
	LONG lResult = recursiveDeleteKey(HKEY_CLASSES_ROOT, szKey);
	assert((lResult == ERROR_SUCCESS) ||
	     (lResult == ERROR_FILE_NOT_FOUND)); // Subkey may not exist.

	// Delete the version-independent ProgID Key.
	lResult = recursiveDeleteKey(HKEY_CLASSES_ROOT, szVerIndProgID);
	assert((lResult == ERROR_SUCCESS) ||
	     (lResult == ERROR_FILE_NOT_FOUND)); // Subkey may not exist.

	// Delete the ProgID key.
	lResult = recursiveDeleteKey(HKEY_CLASSES_ROOT, szProgID);
	assert((lResult == ERROR_SUCCESS) ||
	     (lResult == ERROR_FILE_NOT_FOUND)); // Subkey may not exist.

	return S_OK;
}

///////////////////////////////////////////////////////////
// Internal helper functions

// Convert a CLSID to a char string.
void CLSIDtochar(const CLSID& clsid,
                 char* szCLSID,
                 int length)
{
	assert(length >= CLSID_STRING_SIZE) ;
	// Get CLSID
	LPOLESTR wszCLSID = NULL;
	HRESULT hr = StringFromCLSID(clsid, &wszCLSID);
	assert(SUCCEEDED(hr));

	// Covert from wide characters to non-wide.
	wcstombs(szCLSID, wszCLSID, length);
```

```
        // Free memory.
        CoTaskMemFree(wszCLSID) ;
}

// Delete a key and all of its descendents.
LONG recursiveDeleteKey(HKEY hKeyParent,          // Parent of key to delete
                const char* lpszKeyChild)   // Key to delete
{
        // Open the child.
        HKEY hKeyChild;
        LONG lRes = RegOpenKeyEx(hKeyParent, lpszKeyChild, 0,
                        KEY_ALL_ACCESS, &hKeyChild);
        if (lRes != ERROR_SUCCESS)
        {
                return lRes;
        }

        // Enumerate all of the decendents of this child.
        FILETIME time;
        char szBuffer[256];
        DWORD dwSize = 256;
        while (RegEnumKeyEx(hKeyChild, 0, szBuffer, &dwSize, NULL,
                        NULL, NULL, &time) == S_OK)
        {
                // Delete the decendents of this child.
                lRes = recursiveDeleteKey(hKeyChild, szBuffer);
                if (lRes != ERROR_SUCCESS)
                {
                        // Cleanup before exiting.
                        RegCloseKey(hKeyChild);
                        return lRes;
                }
                dwSize = 256;
        }

        // Close the child.
        RegCloseKey(hKeyChild);

        // Delete this child.
        return RegDeleteKey(hKeyParent, lpszKeyChild);
}

// Create a key and set its value.
BOOL setKeyAndValue(const char* szKey,
                const char* szSubkey,
                const char* szValue)
{
        HKEY hKey;
        char szKeyBuf[1024];

        // Copy keyname into buffer.
        strcpy(szKeyBuf, szKey);

        // Add subkey name to buffer.
        if (szSubkey != NULL)
```

```
	{
		strcat(szKeyBuf, "\\");
		strcat(szKeyBuf, szSubkey );
	}

	// Create and open key and subkey.
	long lResult = RegCreateKeyEx(HKEY_CLASSES_ROOT ,
				szKeyBuf,
				0, NULL, REG_OPTION_NON_VOLATILE,
				KEY_ALL_ACCESS, NULL,
				&hKey, NULL);
	if (lResult != ERROR_SUCCESS)
	{
		return FALSE;
	}

	// Set the Value.
	if (szValue != NULL)
	{
		RegSetValueEx(hKey, NULL, 0, REG_SZ,
			(BYTE *)szValue,
			strlen(szValue)+1 );
	}

	RegCloseKey(hKey);
	return TRUE;
}


==================================================
// InProcServerExample001.def
LIBRARY		InProcServerExample001.dll
DESCRIPTION	an in-proc server example

EXPORTS
	DllGetClassObject	@2	PRIVATE
	DllCanUnloadNow	@3		PRIVATE
	DllRegisterServer	@4	PRIVATE
	DllUnregisterServer @5	PRIVATE


==================================================
// InProcServerComponent.cpp
// Implementation of a in-proc server component

#define INPROCSERVERCOMPONENT_CPP

#include "iostream.h"
#include "objbase.h"

#include "Iface.h"	// Interface declarations
#include "Registry.h"	// Registry helper functions

void trace(const char* msg) { cout << msg << endl ;}

///////////////////////////////////////////////////////////
```

```
// Global variables
static HMODULE g_hModule = NULL; //DLL module handle
static long g_cComponents = 0;  //Count of active components
static long g_cServerLocks = 0;  //Count of locks

// Friendly name of component
const char g_szFriendlyName[] = "First In-Proc Server Example";

// Version-independent ProgID
const char g_szVerIndProgID[] = "InProcServer.InProcServerExample001" ;

// ProgID
const char g_szProgID[] = "InProcServer.InProcServerExample001.1" ;


// Component
class CA : public IX,
        public IY
{
public:
    // interface IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

    // Interface IX
  virtual HRESULT __stdcall Fx() { cout << "Component:\tFx is called!" << endl; return NOERROR; }

    // Interface IY
  virtual HRESULT __stdcall Fy() { cout << "Component:\tFy is called!" << endl; return NOERROR; }

  CA();
  ~CA();

private:
  long m_cRef; //Reference count
};


CA::CA() : m_cRef(1)
{
  InterlockedIncrement(&g_cComponents);
  trace("Component:\tCreating...");
}

CA::~CA()
{
  InterlockedDecrement(&g_cComponents);
  trace("Component:\tDestroy self.");
}


// IUnknown implementation

HRESULT __stdcall CA::QueryInterface
```

```
  (const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
    *ppv = static_cast<IX*>(this);
  else if (iid == IID_IX)
  {
    *ppv = static_cast<IX*>(this);
    trace("Component:\tReturn pointer to IX.");
  }
  else if (iid == IID_IY)
  {
    *ppv = static_cast<IY*>(this);
    trace("Component:\tReturn pointer to IY.");
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

ULONG __stdcall CA::AddRef()
{
  return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CA::Release()
{
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}


///////////////////////////////////////////////////////////
// Class factory

class CFactory : public IClassFactory
{
public:
  // interface IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG   __stdcall AddRef();
  virtual ULONG   __stdcall Release();

  // Interface IClassFactory
  virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv);
  virtual HRESULT __stdcall LockServer(BOOL bLock);
```

```
  CFactory() : m_cRef(1) { trace("Class factory:\tCreating..."); }
  ~CFactory() { trace("Class factory:\tDestroy self."); }

private:
  long m_cRef;
};

// Class factory IUnknown implementation
//
HRESULT __stdcall CFactory::QueryInterface(const IID& iid, void** ppv)
{
  if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
  {
    *ppv = static_cast<IClassFactory*>(this);
    trace("Class factory:\treturning interface...");
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

ULONG __stdcall CFactory::AddRef()
{
  return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CFactory::Release()
{
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}

// IClassFactory implementation

HRESULT __stdcall CFactory::CreateInstance
  (IUnknown* pUnknownOuter,
   const IID& iid,
   void** ppv)
{
  trace("Class factory:\tCreate component.");

  // Cannot aggregate.
  if (pUnknownOuter != NULL)
    return CLASS_E_NOAGGREGATION;

  // Create component.
  CA* pA = new CA;
  if (pA == NULL)
```

```
    return E_OUTOFMEMORY;

  // Get the requested interface.
  HRESULT hr = pA->QueryInterface(iid, ppv);

  // Release the IUnknown pointer.
  // If QueryInterface failed, component will delete itself.
  pA->Release();
  return hr ;
}

// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
  if (bLock)
    InterlockedIncrement(&g_cServerLocks);
  else
    InterlockedDecrement(&g_cServerLocks);
  trace("Class factory:\tLock Server");

  return S_OK ;
}

///////////////////////////////////////////////////////////
//
// Exported functions
//

// Can DLL unload now?
STDAPI DllCanUnloadNow()
{
  trace("DllCanUnloadNow:\tDll can unload now?");
  if ((g_cComponents == 0) && (g_cServerLocks == 0))
    return S_OK;
  else
    return S_FALSE;
}

// Get class factory
STDAPI DllGetClassObject(const CLSID& clsid,
                  const IID& iid,
                  void** ppv)
{
  trace("DllGetClassObject:\tCreate class factory.");

  // Can we create this component?
  if (clsid != CLSID_Component1)
    return CLASS_E_CLASSNOTAVAILABLE ;

  // Create class factory.
  CFactory* pFactory = new CFactory;// No AddRef in constructor
  if (pFactory == NULL)
    return E_OUTOFMEMORY ;

  // Get requested interface.
  HRESULT hr = pFactory->QueryInterface(iid, ppv) ;
  pFactory->Release();
```

```
  return hr;
}

// Server registration
STDAPI DllRegisterServer()
{
  trace("DllRegisterServer:\tRegistering Server...");
  return RegisterServer(g_hModule,
                        CLSID_Component1,
                        g_szFriendlyName,
                        g_szVerIndProgID,
                        g_szProgID);
}

// Server unregistration
STDAPI DllUnregisterServer()
{
  trace("DllUnregisterServer:\tUnregistering server...");
  return UnregisterServer(CLSID_Component1,
                          g_szVerIndProgID,
                          g_szProgID);
}

// DLL module information

BOOL APIENTRY DllMain(HANDLE hModule,
            DWORD dwReason,
            void* lpReserved)
{
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    g_hModule = hModule;
    trace("DLLMain is called for process attachment!");
  }
  else if (dwReason == DLL_PROCESS_DETACH)
    trace("DLLMain is called for process detachment!");

  return TRUE;
}
```

Appendix 10 An In-proc Server Component Client Example

```
================================================
// Client.cpp
// Implementation file of a client which uses an in-proc
// server component.

#include "iostream.h"
#include "objbase.h"
#include "windows.h"

#include "Iface.h"

void trace(const char* msg) { cout << "Client: \t" << msg << endl; }

int main()
{
  // Initialize COM Library
  CoInitialize(NULL);

  trace("Call CoCreateInstance to create component and get interface IX.");
  IX* pIX = NULL;
  HRESULT hr = ::CoCreateInstance(CLSID_Component1,
                       NULL,
                       CLSCTX_INPROC_SERVER,
                       IID_IX,
                       (void**)&pIX);
  if (SUCCEEDED(hr))
  {
    trace("Succeeded getting IX.");
    pIX->Fx(); // Use interface IX.

    trace("Ask for interface IY.");
    IY* pIY = NULL ;
    hr = pIX->QueryInterface(IID_IY, (void**)&pIY);
    if (SUCCEEDED(hr))
    {
      trace("Succeeded getting IY.");
      pIY->Fy(); // Use interface IY.
      pIY->Release();
      trace("Release IY interface.");
    }
    else
      trace("Could not get interface IY.") ;

    trace("Ask for interface IZ.");

    IZ* pIZ = NULL;
    hr = pIX->QueryInterface(IID_IZ, (void**)&pIZ);
    if (SUCCEEDED(hr))
    {
      trace("Succeeded in getting interface IZ.");
      pIZ->Fz();
      pIZ->Release();
```

```
      trace("Release IZ interface."):
    }
    else
      trace("Could not get interface IZ."):

    trace("Release IX interface."):
    pIX->Release():
  }
  else
  {
    cout << "Client: \tCould not create component. hr = "
         << hex << hr << endl:
  }

  // Uninitialize COM Library
  CoUninitialize():

  MessageBox(NULL.
      "When finish viewing the execution result. click OK to terminate!".
      "Message". MB_SYSTEMMODAL):

  return 0:
}
```

Appendix 11 An ActiveX Control Example

```
===============================================
// First.h

#if !defined(AFX_FIRST_H__E660D26D_4115_11D3_B5B1_8876DCED2447__INCLUDED_)
#define AFX_FIRST_H__E660D26D_4115_11D3_B5B1_8876DCED2447__INCLUDED_

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

// First.h : main header file for FIRST.DLL

#if !defined( __AFXCTL_H__ )
        #error include 'afxctl.h' before including this file
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
// CFirstApp : See First.cpp for implementation.

class CFirstApp : public COleControlModule
{
public:
        BOOL InitInstance();
        int ExitInstance();
};

extern const GUID CDECL _tlid;
extern const WORD _wVerMajor;
extern const WORD _wVerMinor;

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

#endif   !defined(AFX_FIRST_H__E660D26D_4115_11D3_B5B1_8876DCED2447__INCLUDED)


===============================================
// First.cpp : Implementation of CFirstApp and DLL registration.

#include "stdafx.h"
#include "First.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


CFirstApp NEAR theApp;
```

```
const GUID CDECL BASED_CODE _tlid =
		{ 0xe660d264, 0x4115, 0x11d3, { 0xb5, 0xb1, 0x88, 0x76, 0xdc, 0xed, 0x24, 0x47 } };
const WORD _wVerMajor = 1;
const WORD _wVerMinor = 0;


////////////////////////////////////////////////////////////////////////////
// CFirstApp::InitInstance - DLL initialization

BOOL CFirstApp::InitInstance()
{
	BOOL bInit = COleControlModule::InitInstance();

	if (bInit)
	{
		// TODO: Add your own module initialization code here.
	}

	return bInit;
}


////////////////////////////////////////////////////////////////////////////
// CFirstApp::ExitInstance - DLL termination

int CFirstApp::ExitInstance()
{
	// TODO: Add your own module termination code here.

	return COleControlModule::ExitInstance();
}


/////////////////////////////////////////////////////////////////////////////
// DllRegisterServer - Adds entries to the system registry

STDAPI DllRegisterServer(void)
{
	AFX_MANAGE_STATE(_afxModuleAddrThis);

	if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
		return ResultFromScode(SELFREG_E_TYPELIB);

	if (!COleObjectFactoryEx::UpdateRegistryAll(TRUE))
		return ResultFromScode(SELFREG_E_CLASS);

	return NOERROR;
}


/////////////////////////////////////////////////////////////////////////////
// DllUnregisterServer - Removes entries from the system registry

STDAPI DllUnregisterServer(void)
{
```

```
	AFX_MANAGE_STATE(_afxModuleAddrThis);

	if (!AfxOleUnregisterTypeLib(_tlid, _wVerMajor, _wVerMinor))
		return ResultFromScode(SELFREG_E_TYPELIB);

	if (!COleObjectFactoryEx::UpdateRegistryAll(FALSE))
		return ResultFromScode(SELFREG_E_CLASS);

	return NOERROR;
}
```

```
==============================================
// FirstCTL.h
#if !defined(AFX_FIRSTCTL_H__E660D275_4115_11D3_B5B1_8876DCED2447__INCLUDED_)
#define AFX_FIRSTCTL_H__E660D275_4115_11D3_B5B1_8876DCED2447__INCLUDED_

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

// FirstCtl.h : Declaration of the CFirstCtrl ActiveX Control class.

/////////////////////////////////////////////////////////////////////////////
// CFirstCtrl : See FirstCtl.cpp for implementation.

class CFirstCtrl : public COleControl
{
	DECLARE_DYNCREATE(CFirstCtrl)

// Constructor
public:
	CFirstCtrl();

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CFirstCtrl)
	public:
	virtual void OnDraw(CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid);
	virtual void DoPropExchange(CPropExchange* pPX);
	virtual void OnResetState();
	//}}AFX_VIRTUAL

// Implementation
protected:
	~CFirstCtrl();

	DECLARE_OLECREATE_EX(CFirstCtrl)    // Class factory and guid
	DECLARE_OLETYPELIB(CFirstCtrl)      // GetTypeInfo
	DECLARE_PROPPAGEIDS(CFirstCtrl)     // Property page IDs
	DECLARE_OLECTLTYPE(CFirstCtrl)		// Type name and misc status

// Message maps
	//{{AFX_MSG(CFirstCtrl)
		// NOTE - ClassWizard will add and remove member functions here.
		//    DO NOT EDIT what you see in these blocks of generated code !
```

```
	}}AFX_MSG
	DECLARE_MESSAGE_MAP()

 Dispatch maps
	{{AFX_DISPATCH(CFirstCtrl)
		NOTE - ClassWizard will add and remove member functions here.
		   DO NOT EDIT what you see in these blocks of generated code !
	}}AFX_DISPATCH
	DECLARE_DISPATCH_MAP()

	afx_msg void AboutBox();

 Event maps
	{{AFX_EVENT(CFirstCtrl)
		NOTE - ClassWizard will add and remove member functions here.
		   DO NOT EDIT what you see in these blocks of generated code !
	}}AFX_EVENT
	DECLARE_EVENT_MAP()

 Dispatch and event IDs
public:
	enum {
	{{AFX_DISP_ID(CFirstCtrl)
		NOTE: ClassWizard will add and remove enumeration elements here.
		   DO NOT EDIT what you see in these blocks of generated code !
	}}AFX_DISP_ID
	};
};

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

#endif  !defined(AFX_FIRSTCTL_H__E660D275_4115_11D3_B5B1_8876DCED2447__INCLUDED)


================================================
 FirstCtl.cpp : Implementation of the CFirstCtrl ActiveX Control class.

#include "stdafx.h"
#include "First.h"
#include "FirstCtl.h"
#include "FirstPpg.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


IMPLEMENT_DYNCREATE(CFirstCtrl, COleControl)


 Message map
```

```
BEGIN_MESSAGE_MAP(CFirstCtrl, COleControl)
	//{{AFX_MSG_MAP(CFirstCtrl)
	// NOTE - ClassWizard will add and remove message map entries
	//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG_MAP
	ON_OLEVERB(AFX_IDS_VERB_PROPERTIES, OnProperties)
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// Dispatch map

BEGIN_DISPATCH_MAP(CFirstCtrl, COleControl)
	//{{AFX_DISPATCH_MAP(CFirstCtrl)
	// NOTE - ClassWizard will add and remove dispatch map entries
	//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_DISPATCH_MAP
	DISP_FUNCTION_ID(CFirstCtrl, "AboutBox", DISPID_ABOUTBOX, AboutBox, VT_EMPTY,
VTS_NONE)
END_DISPATCH_MAP()


/////////////////////////////////////////////////////////////////////////////
// Event map

BEGIN_EVENT_MAP(CFirstCtrl, COleControl)
	//{{AFX_EVENT_MAP(CFirstCtrl)
	// NOTE - ClassWizard will add and remove event map entries
	//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_EVENT_MAP
END_EVENT_MAP()


/////////////////////////////////////////////////////////////////////////////
// Property pages

// TODO: Add more property pages as needed.  Remember to increase the count!
BEGIN_PROPPAGEIDS(CFirstCtrl, 1)
	PROPPAGEID(CFirstPropPage::guid)
END_PROPPAGEIDS(CFirstCtrl)


/////////////////////////////////////////////////////////////////////////////
// Initialize class factory and guid

IMPLEMENT_OLECREATE_EX(CFirstCtrl, "FIRST.FirstCtrl.1",
	0xe660d267, 0x4115, 0x11d3, 0xb5, 0xb1, 0x88, 0x76, 0xdc, 0xed, 0x24, 0x47)


/////////////////////////////////////////////////////////////////////////////
// Type library ID and version

IMPLEMENT_OLETYPELIB(CFirstCtrl, _tlid, _wVerMajor, _wVerMinor)
```

```
// Interface IDs

const IID BASED_CODE IID_DFirst =
		{ 0xe660d265, 0x4115, 0x11d3, { 0xb5, 0xb1, 0x88, 0x76, 0xdc, 0xed, 0x24, 0x47 } };
const IID BASED_CODE IID_DFirstEvents =
		{ 0xe660d266, 0x4115, 0x11d3, { 0xb5, 0xb1, 0x88, 0x76, 0xdc, 0xed, 0x24, 0x47 } };


// Control type information

static const DWORD BASED_CODE _dwFirstOleMisc =
	OLEMISC_ACTIVATEWHENVISIBLE |
	OLEMISC_SETCLIENTSITEFIRST |
	OLEMISC_INSIDEOUT |
	OLEMISC_CANTLINKINSIDE |
	OLEMISC_RECOMPOSEONRESIZE;

IMPLEMENT_OLECTLTYPE(CFirstCtrl, IDS_FIRST, _dwFirstOleMisc)


// CFirstCtrl::CFirstCtrlFactory::UpdateRegistry -
// Adds or removes system registry entries for CFirstCtrl

BOOL CFirstCtrl::CFirstCtrlFactory::UpdateRegistry(BOOL bRegister)
{
	// TODO: Verify that your control follows apartment-model threading rules.
	// Refer to MFC TechNote 64 for more information.
	// If your control does not conform to the apartment-model rules, then
	// you must modify the code below, changing the 6th parameter from
	// afxRegApartmentThreading to 0.

	if (bRegister)
		return AfxOleRegisterControlClass(
			AfxGetInstanceHandle(),
			m_clsid,
			m_lpszProgID,
			IDS_FIRST,
			IDB_FIRST,
			afxRegApartmentThreading,
			_dwFirstOleMisc,
			_tlid,
			_wVerMajor,
			_wVerMinor);
	else
		return AfxOleUnregisterClass(m_clsid, m_lpszProgID);
}


// CFirstCtrl::CFirstCtrl - Constructor

CFirstCtrl::CFirstCtrl()
{
```

```
	InitializeIIDs(&IID_DFirst, &IID_DFirstEvents);

	// TODO: Initialize your control's instance data here.
}


/////////////////////////////////////////////////////////////////////////////
// CFirstCtrl::~CFirstCtrl - Destructor

CFirstCtrl::~CFirstCtrl()
{
	// TODO: Cleanup your control's instance data here.
}


/////////////////////////////////////////////////////////////////////////////
// CFirstCtrl::OnDraw - Drawing function

void CFirstCtrl::OnDraw(
			CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
	// TODO: Replace the following code with your own drawing code.
	pdc->FillRect(rcBounds, CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
	pdc->Ellipse(rcBounds);
}


/////////////////////////////////////////////////////////////////////////////
// CFirstCtrl::DoPropExchange - Persistence support

void CFirstCtrl::DoPropExchange(CPropExchange* pPX)
{
	ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
	COleControl::DoPropExchange(pPX);

	// TODO: Call PX_ functions for each persistent custom property.

}


/////////////////////////////////////////////////////////////////////////////
// CFirstCtrl::OnResetState - Reset control to default state

void CFirstCtrl::OnResetState()
{
	COleControl::OnResetState();  // Resets defaults found in DoPropExchange

	// TODO: Reset any other control state here.
}


/////////////////////////////////////////////////////////////////////////////
// CFirstCtrl::AboutBox - Display an "About" box to the user

void CFirstCtrl::AboutBox()
{
```

```
        CDialog dlgAbout(IDD_ABOUTBOX_FIRST);
        dlgAbout.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
// CFirstCtrl message handlers


==============================================
// FirstPpg.h
#if !defined(AFX_FIRSTPPG_H__E660D277_4115_11D3_B5B1_8876DCED2447__INCLUDED_)
#define AFX_FIRSTPPG_H__E660D277_4115_11D3_B5B1_8876DCED2447__INCLUDED_

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

// FirstPpg.h : Declaration of the CFirstPropPage property page class.

/////////////////////////////////////////////////////////////////////////////
// CFirstPropPage : See FirstPpg.cpp.cpp for implementation.

class CFirstPropPage : public COlePropertyPage
{
        DECLARE_DYNCREATE(CFirstPropPage)
        DECLARE_OLECREATE_EX(CFirstPropPage)

// Constructor
public:
        CFirstPropPage();

// Dialog Data
        //{{AFX_DATA(CFirstPropPage)
        enum { IDD = IDD_PROPPAGE_FIRST };
                // NOTE - ClassWizard will add data members here.
                //    DO NOT EDIT what you see in these blocks of generated code !
        //}}AFX_DATA

// Implementation
protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

// Message maps
protected:
        //{{AFX_MSG(CFirstPropPage)
                // NOTE - ClassWizard will add and remove member functions here.
                //    DO NOT EDIT what you see in these blocks of generated code !
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

};

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

```
#endif // !defined(AFX_FIRSTPPG_H__E660D277_4115_11D3_B5B1_8876DCED2447__INCLUDED)
===================================================
// FirstPpg.cpp : Implementation of the CFirstPropPage property page class.

#include "stdafx.h"
#include "First.h"
#include "FirstPpg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


IMPLEMENT_DYNCREATE(CFirstPropPage, COlePropertyPage)


/////////////////////////////////////////////////////////////////////////////
// Message map

BEGIN_MESSAGE_MAP(CFirstPropPage, COlePropertyPage)
	//{{AFX_MSG_MAP(CFirstPropPage)
	// NOTE - ClassWizard will add and remove message map entries
	//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// Initialize class factory and guid

IMPLEMENT_OLECREATE_EX(CFirstPropPage, "FIRST.FirstPropPage.1",
	0xe660d268, 0x4115, 0x11d3, 0xb5, 0xb1, 0x88, 0x76, 0xdc, 0xed, 0x24, 0x47)


/////////////////////////////////////////////////////////////////////////////
// CFirstPropPage::CFirstPropPageFactory::UpdateRegistry -
// Adds or removes system registry entries for CFirstPropPage

BOOL CFirstPropPage::CFirstPropPageFactory::UpdateRegistry(BOOL bRegister)
{
	if (bRegister)
		return AfxOleRegisterPropertyPageClass(AfxGetInstanceHandle(),
			m_clsid, IDS_FIRST_PPG);
	else
		return AfxOleUnregisterClass(m_clsid, NULL);
}


/////////////////////////////////////////////////////////////////////////////
// CFirstPropPage::CFirstPropPage - Constructor

CFirstPropPage::CFirstPropPage() :
	COlePropertyPage(IDD, IDS_FIRST_PPG_CAPTION)
{
```

```
    {{AFX_DATA_INIT(CFirstPropPage)
    NOTE: ClassWizard will add member initialization here
      DO NOT EDIT what you see in these blocks of generated code !
    }}AFX_DATA_INIT
}

CFirstPropPage::DoDataExchange - Moves data between page and properties

void CFirstPropPage::DoDataExchange(CDataExchange* pDX)
{
    {{AFX_DATA_MAP(CFirstPropPage)
    NOTE: ClassWizard will add DDP, DDX, and DDV calls here
      DO NOT EDIT what you see in these blocks of generated code !
    }}AFX_DATA_MAP
    DDP_PostProcessing(pDX);
}

CFirstPropPage message handlers
```