The Component Object Model

Simplified by: Dr. Vijay Gokhale Sir

Contents

1. Introduction

This chapter explains at a high level the motivations of Component Object Model (COM) and the problems it addresses. It describes what COM is and its features, and describes the major benefits and advantages of COM.

1.1 Current Industry Challenges

With powerful and sophisticated applications have come problems for application developers, software vendors, and users.

1. Applications are large and complex – time consuming to develop, difficult & costly to maintain, risky to extend with additional functionality.
2. Applications are monolithic – application provided features cannot be removed, upgraded independently, or replaced with alternatives.
3. Non-integrated – data and functionality of one application are not readily available to other applications, even if the applications are written in the same programming language and running on the same machine.
4. Programming models are inconsistent for no good reason. Programming models vary widely depending on whether the service is coming from a provider in the same address space as the client program (via dynamic linking), from a separate process on the same machine, from the operating system, or from a provider running on a separate machine across the network.

Additionally, hardware down-sizing and increasing software complexity is the need for a new style of distributed, client/server, modular and "componentized" computing. This style calls for:

1. A generic set of facilities for finding and using service providers (whether provided by the operating system or by applications, or a combination of both). For negotiating capabilities with service providers, and for extending and evolving service providers such a way that does not inadvertently break the consumers of earlier versions of those services.
2. Use object-oriented concepts in system and application service architectures to manage increasing software complexity through increased modularity, to re-use existing solutions and to facilitate more self-sufficient software components.
3. Optimum use of increasingly powerful desktop services, network servers, and legacy systems by clients and servers.
4. Distributed computing to provide a single system image to users and applications and to permit use of services in a networked environments regardless of location, machine architecture, or implementation environment.

1.2 The Solution: Component Software

The solution is a system in which application developers create reusable software component. A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little efforts. For example, a component might be a spelling checker sold by one vendor that can be plugged into several different word processing applications from multiple vendors. Software component must adhere to a binary external standard, but their internal implementation is completely unconstrained.

The Component Object Model (COM) enables software suppliers to package their functions into reusable software components. What COM and its objects do is bring software into the world where an application developer no longer has to write a sorting algorithm, for example. A sorting algorithm can be packaged as a binary object and shipped into a marketplace of component objects. The developer who need a sorting algorithm just uses any sorting object of the required type without worrying about how the sort is implemented. The developer of the sorting object can avoid the hassles and intellectual property concerns of source-code licensing, and devote total energy to providing the best possible binary version of the sorting algorithm. Moreover, the developer can take advantage of COM's ability to provide easy extensibility and innovation beyond standard services as well as robust support for versioning of components, so that a new component works perfectly with software clients expecting to use a previous version. By enabling the development of component software, COM provides a much more productive way to design, build, sell, use, and reuse software.

1.3 Component Technologies

Overl the period, object oriented programming evolved into local component models, such as Java Beans and distributed object technologies, such as the Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI) and the Component Object Model (COM). Object-oriented technologies have had a large impact on industrial software development process. Features of OO methodologies such as encapsulation and abstraction lead to a more flexible and extensible software product. Objectoriented programming, however has failed

to deliver its expectations with respect to productivity gains and in particular the creation of reuse. Component technologies build on this foundation to provide third-party components that isolate and encapsulate specific functionalities and offer them as services in such a way that they can be adapted and reused without having to change them programatically. Components usually consist of multiple objects and thus have a larger granularity. This characteristic enables them to combine functionalities of the objects and offer them as a single service.

Mainstream component models	
Vendors	**Technology**
Microsoft	OLE-COM, ActiveX, COM+, DCOM, .Net Remoting
OLE - Object Linking and Embedding is a proprietary technology developed by Microsoft that allows embedding andlinking to documents and other objects. For developers, it brought OLE Control Extension (OCX), a way to develop and use custom user interface elements. **COM** - Component Object Model (COM) is a binary-interface standard for software components introduced by Microsoft in 1993. It is used to enable inter-process communication and dynamic object creation in a large range of programming languages. COM is the basis for several other Microsoft technologies and frameworks, including OLE, OLE Automation, ActiveX, COM+, DCOM, the Windows shell, DirectX, UMDF and Windows Runtime. **ActiveX** - ActiveX is a standard that enables software components to interact with one another in a networked environment, regardless of the language(s) used to create them. Most World Wide Web (WWW) users will experience ActiveX technology in the form of ActiveX controls, ActiveX documents, and ActiveX scripts. ActiveX is an open integration platform that provides developers, users, and Web producers a fast and easy way to create integrated programs and content for the Internet and Intranets. **COM+** - COM+ can be used to develop enterprise-wide, mission-critical, distributed applications for Windows. COM+ is designed primarily for Microsoft Visual C++ and Microsoft Visual Basic developers. **DCOM** - Distributed Component Object Model (DCOM) is a proprietary Microsoft technology for communication among software components distributed across networked computers. DCOM, which originally was called "Network OLE", extends Microsoft's COM, and provides the communication substrate underMicrosoft's COM+ application server infrastructure. **.Net Remoting** - .NET remoting enables client applications to use objects in other processes on the same computer or on any other computer available on its network. You can also use .NET remoting to communicate with other application domains in the same process. .NET remoting provides an abstract approach to interprocess communication that separates the remotable object from a specific server and client process and from a specific mechanism of communication.	
Java (Sun Microsystems/Oracle)	JavaBeans, RMI (Java Remote Method Invocation), EJB
JavaBeans - In computing based on the Java Platform, JavaBeans are classes that encapsulate many objects into a single object (the bean). They are serializable, have azero-argument constructor, and allow access to properties using getter and setter methods. The name "Bean" was given to encompass this standard, which aims to create reusable software components for Java. **EJB** - Enterprise JavaBeans (EJB) is a managed, server software for modular construction of enterprise software, and one of several Java APIs. EJB is a server-side software component that encapsulates the business logic of an application. The EJB specification is a subset of the Java EE specification. An EJB web container provides a runtime environment for web related software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services. **RMI** - The Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage collection. RMI-IIOP (read: RMI over IIOP) specifically denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.	
OMG (Object Management Group)	CORBA
CORBA - The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) designed to facilitate the communication of systems that are deployed on diverse platforms. CORBA enables collaboration between systems on different operating systems,programming languages, and computing hardware. CORBA uses an object-oriented model although the systems that use CORBA do not have to be object-oriented. CORBA is an example of the distributed object paradigm.	

1.3.1 Local Component Technology

Component models provide a mechanism by which software engineers can develop applications by composing components through their well defined interfaces rather than developing new or changing existing components. By acquiring components from third-party providers the process of development becomes less time consuming and raises the level of implementation abstraction. Moreover using components that have previously been deployed successfully should reduce bugs in the software. Local component technologies can only operate in an intra-process fashion. Therefore all components must reside on the same host at execution time. Two of the main component technologies include JavaBeans of Sun Microsystems and ActiveX of Microsoft, based on their Component Object Model (COM).

1.3.2 Distributed Object Technology

Object middleware enables communication between multiple objects that reside on different networked hosts (distributed objects). The communication primitive between a client object and a server object is a method invocation. Middleware is responsible for providing transparency layers that deal with distributed systems complexities such as location of objects, heterogeneous hardware/software platforms and numerous object implementation programming languages. Shielding the application developer from such complexities results in simpler design and implementation processes. Besides the core task of transparently relaying object invocations, some middleware technologies offer additional services to the application developer. Examples of these services include security, persistence, naming and trading. The three mainstream object middleware technologies used in industry are the Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI) and Microsoft COM. Java RMI is most suitable when the system purely consists of Java objects that need to communicate on a distributed system. The close integration of Microsoft's COM technology with its other products makes COM the best choice for a Microsoft based environment. The CORBA specification is the result of input from a wide range of OMG members, making its implementations the most generally applicable option.

1.3.3 Distributed Component Models

The motivation for distributed component models partly originate from deficiencies present in available OO middleware technologies. Designs and implementations of applications using middleware invariably have a large focus on middleware which can cause a distraction from the main business problem at hand. Moreover experience has shown that integrating existing middleware technologies is cumbersome and often a source of additional complexity. Distributed component technologies combine the characteristics of components with the functionality of middleware systems to provide inter-process communication between components. That is to say components that can communicate across machine boundaries. Enterprise Java Beans (EJB) and the CORBA Component model are both server-side component models used for developing distributed business objects. These are used on the middle-tier application servers that manage the components at run-time and make them available to remote clients. Both EJB and CORBA components operate by providing components with a container in which they can execute. Each component provides an interface used for life-cycle operations such as creation, migration and destruction, as well as the remote interface it supports. Compliant containers all support the same set of interfaces which means that components can freely migrate between different containers at runtime without the need of reconfiguration or recompilation. Containers themselves run on application servers, which offer services offered by the underlying middleware systems such as transactions, security, persistence and notification. The CORBA component model was developed to provide a distributed component model for use with programming languages other than Java and at the same time achieve interoperability with EJB components.

1.4 The Component Software Solution: OLE's COM

History & Evolution:

One of the first methods of inter-process communication in Windows was Dynamic Data Exchange (DDE), first introduced in 1987, that allowed sending and receiving messages in so-called "conversations" between applications.

Antony Williams, Bob Atkinson and Craig Wittenberg were involved in the creation of the COM architecture. Antony Williams distributed two internal papers in Microsoft that embraced the concept of software components: *Object Architecture: Dealing With the Unknown – or – Type Safety in a Dynamically Extensible Class Library* in 1988 and *On Inheritance: What It Means and How To Use It* in 1990. These papers provided the foundation of many of the ideas behind COM.

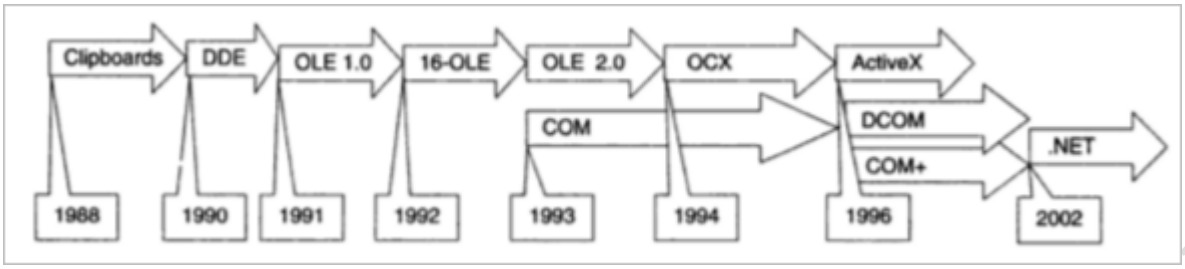


Figure 1.1: OLE and COM Evolution

Object Linking and Embedding (OLE), Microsoft's first object-based framework, was built on top of DDE and designed specifically for compound documents. It was introduced with Word for Windows and Excel in 1991, and was later included with Windows, starting with version 3.1 in 1992. An example of a compound document is a spreadsheet embedded in a Word for Windows document: as changes are made to the spreadsheet within Excel, they appear automatically inside the Word document.

COM is a specification (not a programming language) and a set of services that allow development of modular, object oriented, customizable, upgradable (loosely coupled) and distributed (location transparent) application by number of programming languages (language neutral). COM is an open standard, fully and publicly documented from the lowest levels of its protocols to the highest. The Component Object Model is an object-based programming model designed to promote software interoperability; that is, to allow two or more applications or "components" to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems. To support its interoperability features, COM defines and implements mechanisms that allow applications to connect to each other as software objects. A software object is a collection of related function (or intelligence) and function's (or intelligence's) associated state information.

In other words, COM, like a traditional system-service APIs, provides the operations through which a client of some service can connect to multiple providers of that service in a polymorphic fashion. However, once a connection is established, COM drops out of the picture. COM serves to connect a client and an object, but once that connection is established, the client and object communicate directly without having to suffer overhead of being forced through a central piece of API code as illustrated below.

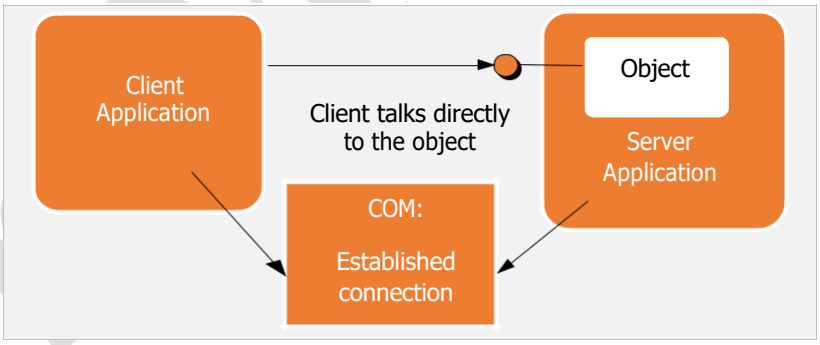


Figure 1.2: Once COM connects client and object, the client and the object communicate directly without added overhead.

COM is not a prescribed way to structure an application; rather, it is a set of technologies for building robust groups of services in both systems and applications such that the services and the clients of those services can evolve over time. In this way, COM is a technology that makes the programming, use, and uncoordinated/independent evolution of binary objects *possible*. COM provides a ready base for extensions oriented towards increased ease-of-use, as well as a great basis for powerful, easy development environments, language-specific improvements to provide better language integration, and pre-packaged functionality within the context of application frameworks.

COM solves the "deployment problem," the versioning/evolution problem where it is necessary that the functionality of objects can incrementally evolve or change without the need to simultaneously change all existing the clients of the object. Objects/services can easily continue to support the interfaces through which they communicated with older clients as well as provide new and better interfaces through which they communicate with newer clients.

To solve the versioning problems as well providing connection services without undue overhead, the Component Object Model builds a foundation that:

Enables the creation and use of reusable components by making them "component objects."
Defines a binary standard for interoperability.
Is a true system object model.
Provides distributed capabilities.

1.4.1 Reusable Component Object

Object-oriented programming allows programmers to build flexible and powerful software objects that can easily be reused by other programmers. The definition of an object is a piece of software that contains the functions that represent what the object can do (its intelligence) and associated state information for those functions (data). An object is, in other words, some data structure and some functions to manipulate that structure.

An important principle of object-oriented programming is encapsulation, where the exact implementation of those functions and the exact format and layout of the data is only of concern to the object itself. This information is hidden from the clients of an object. Those clients are interested only in an object's behaviour and not the object's internals. For instance, consider an object that represents a stack: a user of the stack cares only that the object supports "push" and "pop" operations, not whether the stack is implemented with an array or a linked list. Put another way, a client of an object is interested only in the "contract" - the promised behaviour - that the object supports, not the implementation it uses to fulfil that contract.

Object-Oriented Programming model –

```
Polymorphism + (Some) Late binding + (Some syntactical) Encapsulation +
Inheritance
```

COM programming model –

```
Polymorphism + Real Late binding + Real Binary Encapsulation + Interface
Inheritance + Binary Reusability
```

1.4.2 Binary and Wire-Level Standards for Interoperability

The Component Object Model defines a completely standardized mechanism for creating objects and for clients and objects to communicate. These mechanisms are independent of the applications that use object services and of the programming languages used to create the objects. The mechanisms also support object invocations across the network. COM therefore defines a *binary interoperability standard* rather than a language-based interoperability standard on any given operating system and hardware platform. In the domain of network computing, COM defines a standard architecture-independent wire format and protocol for interaction between objects on heterogeneous platforms.

By providing a binary and network standard, COM enables interoperability among applications that different programmers from different companies write. Without a binary and network standard for communication and a standard set of communication interfaces, programmers face the daunting task of writing a large number of procedures, each of which is specialized for communicating with a different type of object or client, or perhaps recompiling their code depending on the other components or network services with which they need to interact. With a binary and network standard, objects and their clients need no special code and no recompilation for interoperability.

With COM, applications interact with each other and with the system through collections of function calls—also known as methods or member functions or requests—called interfaces. An "interface" in the COM sense is a strongly typed contract between software components to provide a relatively small but useful set of semantically related operations. An interface is an articulation of an expected behaviour and expected responsibilities, and the semantic relation of interfaces gives programmers and designers a concrete entity to use when referring to the contract. Although not a strict requirement of the model, interfaces should be factored in such fashion that they can be re-used in a variety of contexts. For example, a simple interface for generically reading and writing streams of data can be re-used by many different types of objects and clients.

The use of such interfaces in COM provides four major benefits:

1. The ability for functionality in applications (clients or servers of objects) to evolve over time: revising an object by adding new, even unrelated functionality will not require any recompilation on the part of any existing clients. Because COM allows objects to have multiple interfaces, an object can express any number of "versions" simultaneously, each of which may be in simultaneous use by clients of different vintage.
2. Very fast and simple object interaction for same-process objects: Once a client establishes a connection to an object, calls to that object's services (interface functions) are simply indirect functions calls through two memory pointers. As a result, the performance overhead of interacting with an in-process COM object (an object that is in the same address space) as the calling code is negligible
3. Location transparency: The binary standard allows COM to intercept an interface call to an object and make instead a remote procedure call (RPC) to the "real" instance of the object that is running in another process or on another machine. Thought overheads are involved in making a remote procedure call, no special code is necessary in the client to differentiate an in-process object from out-of-process objects. All objects are available to clients in a uniform, transparent fashion.
4. Programming language independence: Because COM is a binary standard, objects can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers—languages such as C, C++, Pascal, Ada, Smalltalk, and even BASIC programming environments—can create and use COM objects immediately.

In sum, only with a binary standard can an object model provide the type of structure necessary for full interoperability, evolution, and re-use between any application or component supplied by any vendor on a single machine architecture. Only with an architecture-independent network wire protocol standard can an object model provide full interoperability, evolution, and re-use between any application or component supplied by any vendor in a network of heterogeneous computers.

1.4.3 A True System Object Model

In addition to being an object-based service architecture, COM is a true system object model because it:

Uses "globally unique identifiers" to identify object classes and the interfaces those objects may support.
Provides methods for code reusability without the problems of traditional language-style implementation inheritance.
Has a single programming model for in-process, cross-process, and cross-network interaction of software components.
Encapsulates the life-cycle of objects via reference counting.
Provides a flexible foundation for security at the object level.

1.4.4 Globally Unique Identifier

Distributed object systems have potentially millions of interfaces and software components that need to be uniquely identified. Any system that uses human-readable names for finding and binding to modules, objects, classes, or requests is at risk because the probability of a collision between human-readable names is nearly 100% in a complex system. The result of name-based identification will inevitably be the accidental connection of two or more software components that were not designed to interact with each other, and a resulting error or crash - even though the components and system had no bugs and worked as designed.

By contrast, COM uses globally unique identifiers (GUIDs) - 128-bit integers that are virtually guaranteed to be unique in the world across space and time - to identify every interface and every object class and type. These globally unique identifiers are the same as UUIDs (Universally Unique IDs) as defined by DCE (Distributed Computing Environment). Human-readable names are assigned only for convenience and are locally scoped. This helps insure that COM components do not accidentally connect to an object or via an interface or method, even in networks with millions of objects. One could generate 10 million GUIDs a second until the year 5770 AD and each one would be unique.

1.4.5 Code Reusability and Implementation Inheritance

Implementation inheritance—the ability of one component to "subclass" or "inherit" some of its functionality from another component while "over-riding" other functions—is a very useful technology for building applications. But more and more experts are concluding that it creates serious problems in a loosely coupled, decentralized, evolving object system. The problem is technically known as the lack of type-safety in the specialization interface. Today, COM provides two mechanisms for code reuse called containment/delegation and aggregation. In the first and more common mechanism, one object (the "outer" object) simply becomes the client of another, internally using the

second object (the "inner" object) as a provider of services that the outer object finds useful in its own implementation. With aggregation, the second and rarer reuse mechanism, COM objects take advantage of the fact that they can support multiple interfaces. An aggregated object is essentially a composite object in which the outer object exposes an interface from the inner object directly to clients as if it were part of the outer object. Both these reuse mechanisms allow objects to exploit existing implementation while avoiding the problems of traditional implementation inheritance. We will see more details on containment and aggregation in the later chapters.

1.4.6 Life-cycle encapsulation

In traditional object systems, the life-cycle of objects—the issues surrounding the creation and deletion of objects—is handled implicitly by the language (or the language runtime) or explicitly by application programmers. In other words, an object-based application, there is always someone (a programmer or team of programmers) or something (for example, the start-up and shutdown code of a language runtime) that has complete knowledge when objects must be created and when they should be deleted.

But in an evolving, decentralized *system* made up of objects, it is no longer true that someone or something always "knows" how to deal with object life-cycle. The solution that is embraced by COM is, clients must tell an object when they are using it and when they are done, and objects must delete themselves when they are no longer needed. This approach, based on reference counting by all objects, is summarized by the phrase "life-cycle encapsulation" since objects are truly encapsulated and self-reliant if and only if they are responsible, with the appropriate help of their clients acting singly and not collectively, for deleting themselves. Reference counting is admittedly complex but its inevitability for a true system object model with full location transparency is apparent.

1.4.7 Security

For a distributed object system to be useful in the real world it must provide a means for secure access to objects and the data they encapsulate.

COM provides security along several crucial dimensions. First, COM uses standard operating system permissions to determine whether a client (running in a particular user's security context) has the right to start the code associated with a particular class of object. Second, with respect to persistent objects (class code along with data stored in a persistent store such as file system or database), COM uses operating system or application permissions to determine if a particular client can load the object at all, and if so whether they have read-only or read-write access, etc. Finally, because its security architecture is based the design of the DCE RPC security architecture, an industry-standard communications mechanism that includes fully authenticated sessions, COM provides cross-process and cross-network object servers with standard security information about the client or clients that are using it so that a server can use security in more sophisticated fashion than that of simple OS permissions on code execution and read/write access to persistent data.

1.4.8 Distributed Capabilities

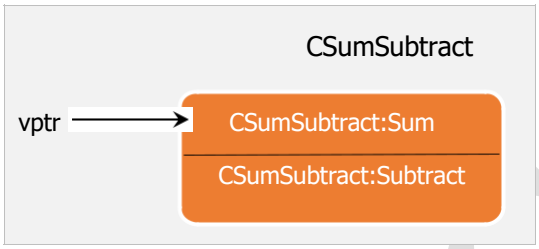
COM supports distributed objects; that is, it allows application developers to split a single application into a number of different component objects, each of which can run on a different computer. Since COM provides network transparency, these applications do not appear to be located on different machines. The entire network appears to be one large computer with enormous processing power and capacity. The Component Object Model provides just such a transparent model, where a client uses an object in the same process in precisely the same manner as it would use one on a machine thousands of miles away. COM explicitly bars certain kinds of "features"—such as direct access to object data, properties, or variables—that might be convenient in the case of in-process objects but would make it impossible for an out-of-process object to provide the same set of services. This is called location transparency.

1.5 Polymorphism and Dynamic Dispatching

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time. Dynamic dispatch contrasts with static dispatch, in which the implementation of a polymorphic operation is selected at compile-time. The purpose of dynamic dispatch is to support cases where the appropriate implementation of a polymorphic operation cannot be determined at compile time. Dynamic dispatch is different from late binding (also known as dynamic binding). In the context of selecting an operation, binding associates a name to an operation. Dispatching chooses an implementation for the operation after you have decided which operation a name refers to. With dynamic dispatch, the name may be bound to a polymorphic operation at compile time, but the implementation not be chosen until run time. While dynamic dispatch does not imply late binding, late binding does imply dynamic dispatching since the binding is what determines the set of available dispatches.

The runtime implementation of virtual functions in C++ takes the form of vptrs and vtbls in virtuall all production compilers. This technique is based on the compiler silently generating a static array of function pointers for each class that contains virtual functions. The array is called the virtual function table (or vtbl) and contains one function pointer for each virtual function defined the class or its base class. Each instance of the class contains a single invisible data member called the virtual function pointer (vptr) that is automatically initialized by the constructor to point to the class's vtbl.

When a client calls a virtual function, the compiler generates the code to dereference the vptr, index into the vtbl, and call through the function pointer found at the designed location. This is how polymorphism and dynamic call dispatching are implemented in C++.



1.6 Objects and Interfaces

An object is an instantiation of some class. At a generic level, a "class" is the definition of a set of related data and capabilities grouped together for some distinguishable common purpose. The purpose is generally to provide some service to "things" outside the object, namely clients that want to make use of those services.

An object that conforms to COM is a special manifestation of this definition of object. A COM object appears in memory much like a C++ object. Unlike C++ objects, however, a client never has direct access to the COM object in its entirety. Instead, clients always access the object through clearly defined contracts: the interfaces that the object supports, and only those interfaces. Thus you can quote, COM is a better C++.

An interface is a strongly-typed group of semantically-related functions, also called "interface member functions." The name of an interface is always prefixed with an "I" by convention, as in IUnknown. (The real identity of an interface is given by its GUID; names are a programming convenience, and the COM system itself uses the GUIDs exclusively when operating on interfaces.) In addition, while the interface has a specific name (or type) and names of member functions, it defines only how one would use that interface and what behaviour is expected from an object through that interface. Interfaces do not define any implementation. For example, a hypothetical interface called IStack that had member functions of Push and Pop would only define the parameters and return types for those functions and what they are expected to do from a client perspective; the object is free to implement the interface as it sees fit, using an array, linked list, or whatever other programming methods it desires.

When an object "implements an interface" that object implements each member function of the interface and provides pointers to those functions to COM. COM then makes those functions available to any client who asks. Syntactically, interface is nothing but a static array of function pointers called *VTable*. This VTable is populated at runtime with the actual virtual methods addresses in the memory and a VTable pointer is used to reference and dereference the virtual methods.

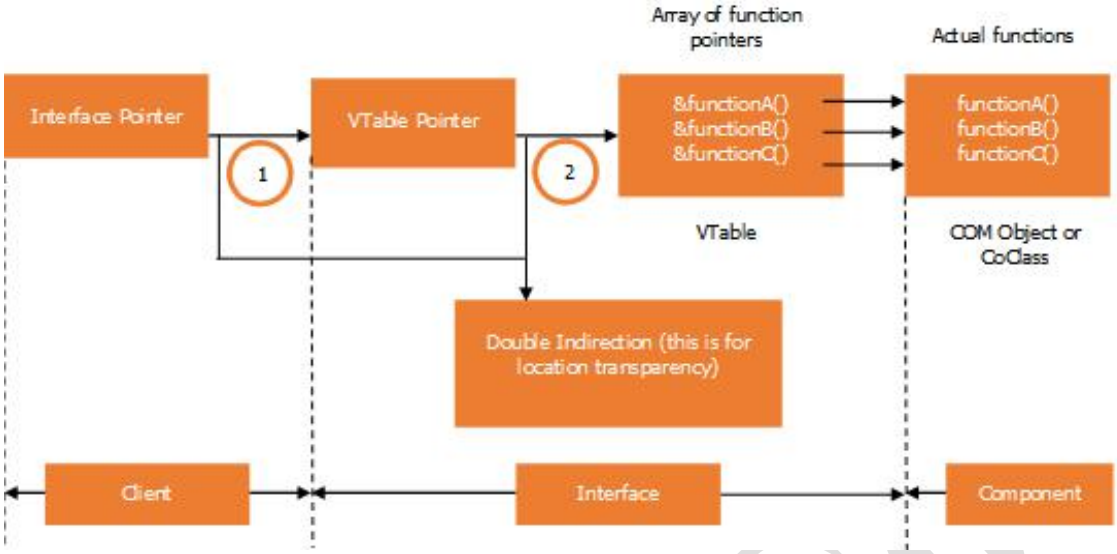


Figure 1.3: Illustrates how Interface methods implemented by a COM Object are accessed using VTABLE

1.6.1 Attributes of Interfaces

Given that an interface is a contractual way for an object to expose its services, there are four very important points to understand:

1. **An interface is not a class:** An interface is not a class in the normal definition of "class." A class can be instantiated to form an object. An interface cannot be instantiated by itself because it carries no implementation. An object must implement that interface and that object must be instantiated for there to be an interface. Furthermore, different object classes may implement an interface differently yet be used interchangeably in binary form, so long as the behaviour conforms to the interface definition (such as two objects that implement IStack where one uses an array and the other a linked list).

2. **An interface is not an object:** An interface is just a related group of functions and is the binary standard through which clients and objects communicate. The object can be implemented in any language with any internal state representation, so long as it can provide pointers to interface member functions.

   2.1 Clients only interact with pointers to interfaces: When a client has access to an object, it has nothing more than a pointer through which it can access the functions in the interface, called simply an interface pointer. The pointer is opaque, meaning that it hides all aspects of internal implementation. You cannot see any details about the object such as its state information, as opposed to C++ object pointers through which a client may directly access the object's data. In COM, the client can only call functions of the interface to which it has a pointer. But instead of being a restriction, this is what allows COM to provide the efficient binary standard that enables location transparency.

   2.2 Objects can implement multiple interfaces: An object class can - and typically does - implement more than one interface. That is, the class has more than one set of services to provide from each object. For example, a class might support the ability to exchange data with clients as well as the ability to save its persistent state information (the data it would need to reload to return to its current state) into a file at the client's request. Each of these abilities is ex-pressed through a different interface, so the object must implement two interfaces.

3. **Interfaces are strongly typed:** Every interface has its own interface identifier (a GUID) thereby eliminating any chance of collision that would occur with human-readable names.

4. **Interfaces are immutable:** Interfaces are never versioned, thus avoiding versioning problems. A new version of an interface, created by adding or removing functions or changing semantics, is an entirely new interface and is assigned a new unique identifier. Therefore a new interface does not conflict with an old interface even if all that changed is the semantics. Objects can, of course, support multiple interfaces simultaneous; and they can have a single internal implementation of the common capabilities exposed through two or more similar interfaces,

such as "versions" (progressive revisions) of an interface. This approach of immutable interfaces and multiple interfaces per object avoids versioning problems.

The encapsulation of functionality into objects accessed through interfaces makes COM an open, extensible system. It is open in the sense that anyone can provide an implementation of a defined interface and anyone can develop an application that uses such interfaces, such as a compound document application. It is extensible in the sense that new or extended interfaces can be defined without changing existing applications and those applications that understand the new interfaces can exploit them while continuing to interoperate with older applications through the old interfaces.

1.6.2 Object Representations (Lollipop Diagrams)

The adopted convention for the standard pictorial representation for objects is to draw each interface on an object as a "plug-in jack." These interfaces are generally drawn out the left or right side of a box representing the object as a whole as illustrated below. If desired, the names of the interfaces are positioned next to the interface jack itself.

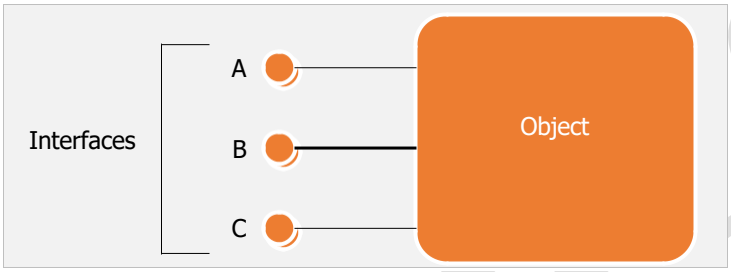


Figure 1.4: A typical picture of an object that supports three interfaces A, B, and C.

If there is no client in the picture then the convention is for interfaces to extend to the left as shown above. With a client in the picture, the interfaces extend towards the client, and the client is understood to have a pointer to one or more of the interfaces on that object as illustrated below.

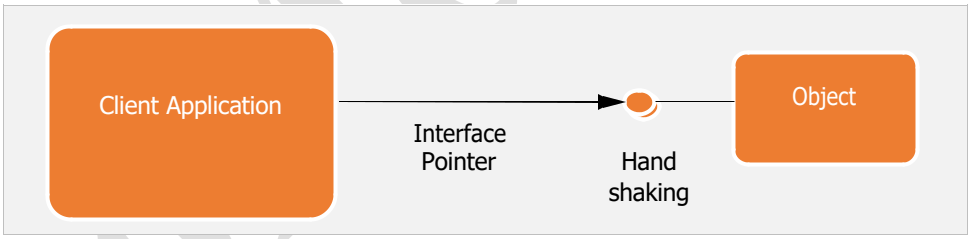


Figure 1.5: Interfaces extend towards the clients connected to them.

In some circumstances a client may itself implement a small object to provide another object with functions to call on various events or to expose services itself. In such cases the client is also an object implementer and the object is also a client.

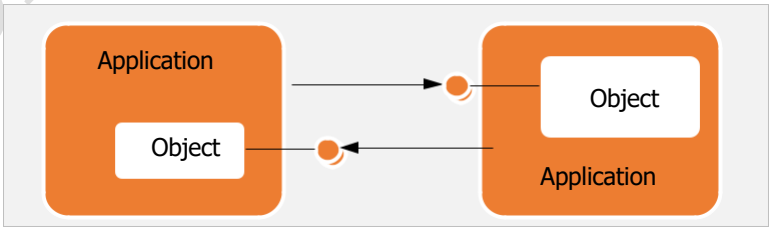
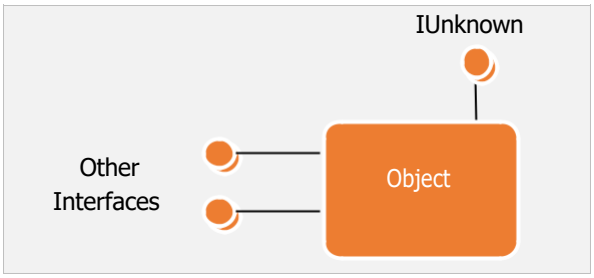


Figure 1.6: Two applications may connect to each other's objects, in which case they extend their interfaces towards each other.

There is one interface that demands a little special attention: IUnknown. This is the base interface of all other interfaces in COM that all objects must support. Usually by implementing any interface at all an object also implements a set of IUnknown functions that are contained within that implemented interface. In some cases, however, an object will implement IUnknown by itself, in which case that interface is extended from the top of the object as shown below.



Fiigure 1.7: The IUnknown interface extends from the top of objects by convention.

In order to use an interface on a object, a client needs to know what it would want to do with that interface—that's what makes it a client of an interface rather than just a client of the object. In the "plug-in jack" concept, a client has to have the right kind of plug to fit into the interface jack in order to do anything with the object through the interface.

1.6.3 Objects with Multiple Interfaces

In COM, an object can support multiple interfaces, that is, provide pointers to more than one grouping of functions. Multiple interfaces is a fundamental innovation of COM as the ability for such avoids versioning problems (interfaces are immutable as described earlier) and any strong association between an interface and an object class. Multiple interfaces is a great improvement over systems in which each object only has one massive interface, and that interface is a collection of everything the object does. Therefore the identity of the object is strongly tied to the exact interface, which introduces the versioning problems once again. Multiple interfaces is the cleanest way around the issue altogether.

1.7 Clients, Servers, and Object Implementers

The interaction between objects and the users of those objects in COM is based on a client/server model. The term 'client' to refer to some piece of code that is using the services of an object. Because an object supplies services, the implementer of that object is usually called the "server," the one who serves those capabilities. A client/server architecture in any computing environment leads to greater robustness: if a server process crashes or is otherwise disconnected from a client, the client can handle that problem gracefully and even restart the server if necessary. As robustness is a primary goal in COM, then a client/server model naturally fits.

However, there is more to COM than just clients and servers. There are also *object implementers*, or some program structure that implements an object of some kind with one or more interfaces on that object. Sometimes a client wishes to provide a mechanism for an object to call back to the client when specific events occur. In such cases, COM specifies that the client itself implements an object and hands that object's first interface pointer to the other object outside the client. In that sense, both sides are clients, both sides are servers in some way. Since this can lead to confusion, the term "server" is applied in a much more specific fashion leading to the following definitions that apply in all of COM:

Object: A unit of functionality that implements one or more interfaces to expose that functionality. For convenience, the word is used both to refer to an object class as well as an individual instantiation of a class. Note that an object class does not need a class identifier in the COM sense such that other applications can instantiate objects of that class—the class used to implement the object internally has no bearing on the externally visible COM class identifier.

Object Implementer: Any piece of code, such as an application, that has implemented an object with any interfaces for any reason. The object is simply a means to expose functions outside the particular application such that outside agents can call those functions. Use of "object" by itself implies an object found in some "object implementer" unless stated otherwise.

In COM world, every COM Object (CoClass) directly or indirectly inherits from the IUnknown interface and provides concrete implementation for its virtual methods.

Direct Inheritance:-

```
Class ISort : public IUnknown // 00000000-0000-0000-C000-000000000046
```

Indirect Inheritance:-

```
Class ISort : public IClassfactory // 00000001-0000-0000-C000-000000000046
```

COM Object (CoClass) = `C++ Object + IUnknown`

Client: There are two definitions of this word. The general definition is any piece of code that is using the services of some object, wherever that object might be implemented. A client of this sort is also called an "object user." The second definition is the active agent (an application) that drives the flow of operation between itself and other objects and uses specific COM "implementation locator" services to instantiate or create objects through servers of various object classes.

Server: A piece of code that structures an object class in a specific fashion and assigns that class a COM class identifier. This enables a client to pass the class identifier to COM and ask for an object of that class. COM is able to load and run the server code, ask the sever to create an object of the class, and connect that new object to the client. A server is specifically the necessary structure around an object that serves the object to the rest of the system and associates the class identifier: a server is not the object itself. The word "server" is used in discussions to emphasize the serving agent more than the object. The phrase "server object" is used specifically to identify an object that is implemented in a server when the context is appropriate.

Putting all of these pieces together, imagine a client application that initially uses COM services to create an object of a particular class. COM will run the server associated with that class and have it create an object, returning an interface pointer to the client. With that interface pointer the client can query for any other interface on the object. If a client wants to be notified of events that happen in the object in the server, such as a data change, the client itself will implement an "event sink" object and pass the interface pointer to that sink to the server's object through an interface function call. The server holds onto that interface pointer and thus itself becomes a client of the sink object. When the server object detects an appropriate event, it calls the sink object's interface function for that even. The overall configuration created in this scenario is much like that shown earlier in Figure 1.6. There are two primary modules of code (the original client and the server) who both implement objects and who both act in some aspects as clients to establish the configuration.

When both sides in a configuration implement objects then the definition of "client" is usually the second one meaning the active agent who drives the flow of operation between all objects, even when there is more than one piece of code that is acting like a client of the first definition. This specification endeavours to provide enough context to make it clear what code is responsible for what services and operations.

1.7.1 Server Flavours: In-Process and Out-Of-Process

As defined in the last section, a "server" in general is some piece of code that structures some object in such a way that COM "implementer locator" services can run that code and have it create objects. Any specific server can be implemented in one of a number of flavours depending on the structure of the code module and its relationship to the client process that will be using it. A server is either "in-process" which means it's code executes in the same process space as the client, or "out-of-process" which means it runs in another process on the same machine or in another process on a remote machine. These three types of servers are called "in-process," "local," and "remote" as defined below:

In-Process Server: A server that can be loaded into the client's process space and serves "in-process objects." Under Microsoft Windows and Microsoft Windows NT, these are implemented as "dynamic link libraries" or DLLs.

Local Server: A server that runs in a separate process on the same machine as the client and serves "local objects." This type of server is another complete application of its own thus defining the separate process. This specification uses the terms "EXE" or "executable" to describe an application that runs in its own process as opposed to a DLL which must be loaded into an existing process.

Remote Server: A server that runs on a separate machine and therefore always runs in another process as well to serve "remote objects." Remote servers may be implemented in either DLLs or EXEs; if a remote server is implemented in a DLL, a surrogate process will be created for it on the remote machine.

1.7.2 Location Transparency

COM is designed to allow clients to transparently communicate with objects regardless of where those objects are running, be it the same process, the same machine, or a different machine. What this means is that there is a single

programming model for all types of objects for not only clients of those objects but also for the servers of those objects.

From a client's point of view, all objects are access through interface pointers. A pointer must be in-process, and in fact, any call to an interface function always reaches some piece of in-process code first. If the object is in-process, the call reaches it directly, with no intervening system-infrastructure code. If the object is out-of-process, then the call first reaches what is called a "proxy" object provided by COM itself which generates the appropriate remote procedure call to the other process or the other machine.

From a server's point of view, all calls to an object's interface functions are made through a pointer to that interface. Again, a pointer only has context in a single process, and so the caller must always be some piece of in-process code. If the object is in-process, the caller is the client itself. Otherwise, the caller is a "stub" object provided by COM that picks up the remote procedure call from the "proxy" in the client process and turns it into an interface call to the server object.

As far as both clients and servers know, they always communicate directly with some other in-process code as illustrated in Figure 1.8.

The bottom line is that dealing with in-process or remote objects is transparent and identical to dealing with in-process objects. This location transparency has a number of key benefits:

A common solution to problems that are independent of the distance between client and server Programmers leverage their learning
Systems implementation is centralized
Interface designers focus on design

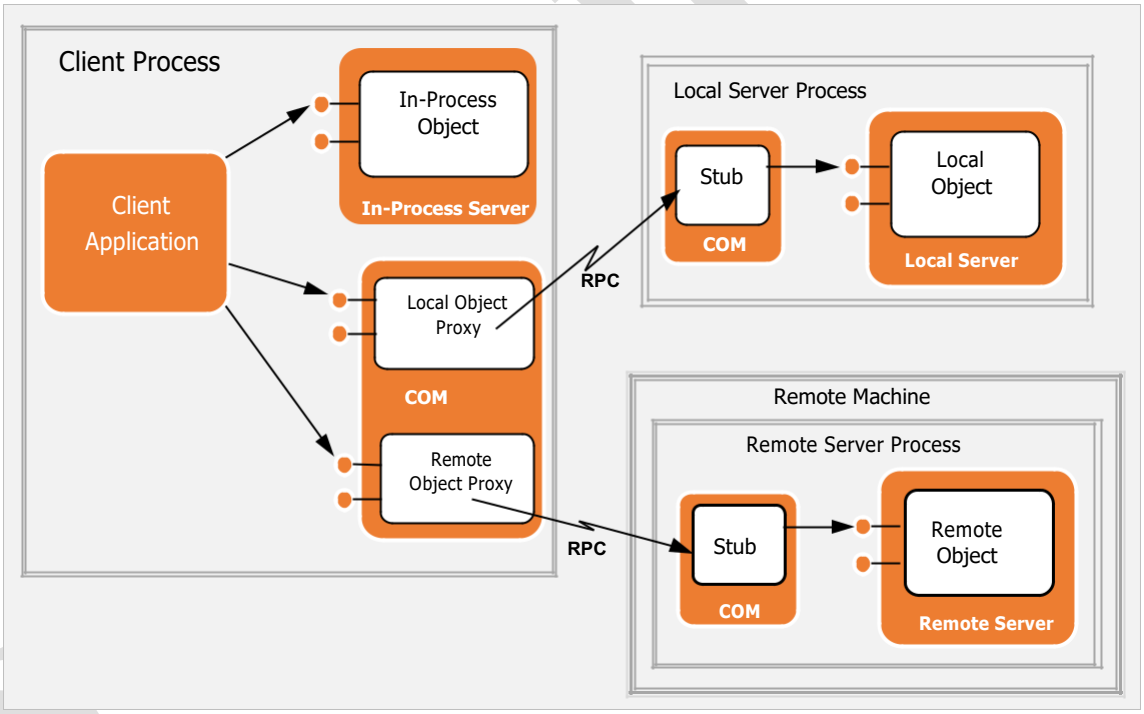


Figure 1.8: Clients always call in-process code; objects are always called by in-process code. COM provides the underlying transparent RPC.

An object implementer can if he wishes support custom marshalling which allows his objects to take special action when they are used from across the network, different action if he would like than is used in the local case. The key point is that this is done completely transparently to the client. Taken as a whole, this architecture allows one to design client / object interfaces at their natural and easy semantic level without regard to network performance issues, then at a later address network performance issues without disrupting the established design.

Also note again that COM is not a specification for how applications are structured: it is a specification for how applications interoperate. For this reason, COM is not concerned with the internal structure of an application—that is the job of programming languages and development environments. Conversely, programming environments have no

set standards for working with objects outside of the immediate application. C++, for example, works extremely well to work with objects inside an application, but has no support for working with objects outside the application. Thus, COM proves to be better C++. Generally all other programming languages are the same in this regard. Therefore COM, through language-independent interfaces, picks up where programming languages leave off to provide the network-wide interoperability.

1.8 The COM Library

COM itself involves some systems-level code, that is, some implementation of its own. However, at the core the Component Object Model by itself is a specification (hence "Model") for how objects and their clients interact through the binary standard of interfaces. As a specification it defines a number of other standards for interoperability:

The fundamental process of interface negotiation through QueryInterface.
A *reference counting* mechanism through objects (and their resources) are managed even when connected to multiple clients.
Rules for memory allocation and responsibility for those allocations when exchanged between independently developed components.
Consistent and rich error reporting facilities.

In addition to being a specification, COM is also an implementation contained what is called the "COM Library." The implementation is provided through a library (such as a DLL on Microsoft Windows) that includes:

A small number of fundamental API functions that facilitate the creation of COM applications, both clients and servers. For clients, COM supplies basic object creation functions; for servers the facilities to expose their objects.
Implementation locator services through which COM determines from a class identifier which server implements that class and where that server is located. This includes support for a level of indirection, usually a system registry, between the identity of an object class and the packaging of the implementation such that clients are independent of the packaging which can change in the future.
Transparent remote procedure calls when an object is running in a local or remote server, as illustrated in Figure 1-8 in the previous section.
A standard mechanism to allow an application to control how memory is allocated within its process.

1.9 COM as a Foundation

The binary standard of interfaces is the key to COM's extensible architecture, providing the foundation upon which is built the rest of COM and other systems such as OLE.

1.9.1 COM Infrastructure

COM provides more than just the fundamental object creation and management facilities: it also builds an infrastructure of three other core operating system components.

Persistent Storage: A set of interfaces and an implementation of those interfaces that create structured storage, otherwise known as a "file system within a file." Information in a file is structured in a hierarchical fashion which enables sharing storage between processes, incremental access to information, transactioning support, and the ability for any code in the system to browse the elements of information in the file. In addition, COM defines standard "persistent storage" interfaces that objects implement to support the ability to save their persistent state to permanent, or persistent, storage devices such that the state of the object can be restored at a later time.

Persistent, Intelligent Names (Monikers): The ability to give a specific instantiation of an object a particular name that would allow a client to reconnect to that exact same object instance with the same state (not just another object of the same class) at a later time. This also includes the ability to assign a name to some sort of operation, such as a query, that could be repeatedly executed using only that name to refer to the operation. This level of indirection allows changes to happen behind the name without requiring any changes to the client that stores that particular name. This technology is centred around a type of object called a moniker and COM defines a set of interfaces that moniker objects implement. COM also defines a standard composite moniker that is used to create complex names that are built of simpler monikers. Monikers also implement one of the persistent storage interfaces meaning that they know how to save their name or other information to somewhere permanent. Monikers are "intelligent" because they know how to take the name information and somehow relocate the specific object or perform an operation to which that name refers.

Uniform Data Transfer: Standard interfaces through which data is exchanged between a client and an object and through which a client can ask an object to send notification (call event functions in the client) in case of a data change. The standards include powerful structures used to describe data formats as well as the storage mediums on which the data is exchanged.

The combination of the foundation and the infrastructure COM components reveals a system that describes how to create and communicate with objects, how to store them, how to label to them, and how to exchange data with them. These four aspects of COM form the core of information management. Furthermore, the infrastructure components not only build on the foundation, but monikers and uniform data transfer also build on storage as shown in Figure 1.9. The result is a system that is not only very rich, but also deep, which means that work done in an application to implement lower level features is leveraged to build higher level features.

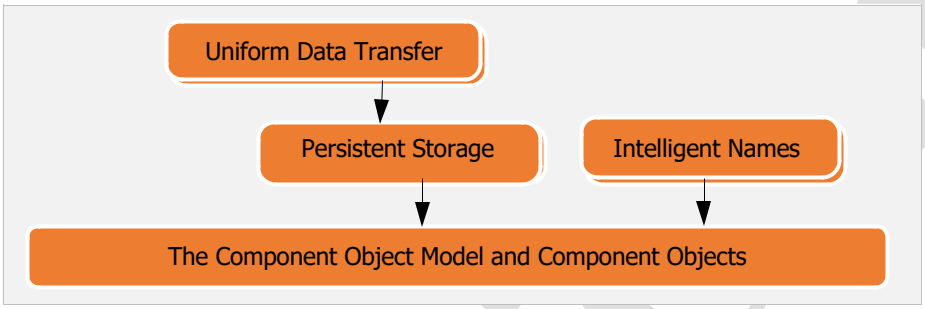


Figure 1.9: COM is built in progressively higher level technologies that depend upon lower level technologies.

1.9.2 Object Linking and Embedding (OLE)

Microsoft's OLE technology is really a collection of additional higher-level technologies that build upon COM and its infrastructure. OLE version 2.0 was the first deployment of a subset of this COM specification that included support for in-process and local objects and all the infrastructure technologies but did not support remote objects

Drag & Drop: The ability to exchange data by picking up a selection with the mouse and visibly dropping it onto another window.

Automation: The ability to create "programmable" applications that can be driven externally from a script running in another application to automate common end user tasks. Automation enables cross-application macro programming.

Compound Documents: The ability to embed or link information in a central document encouraging a more document-centric user interface. Also includes In-Place Activation (also called "Visual Editing") as a user interface improvement to embedding where the end user can works on information from different applications in the context of the compound document without having to switch to other windows.

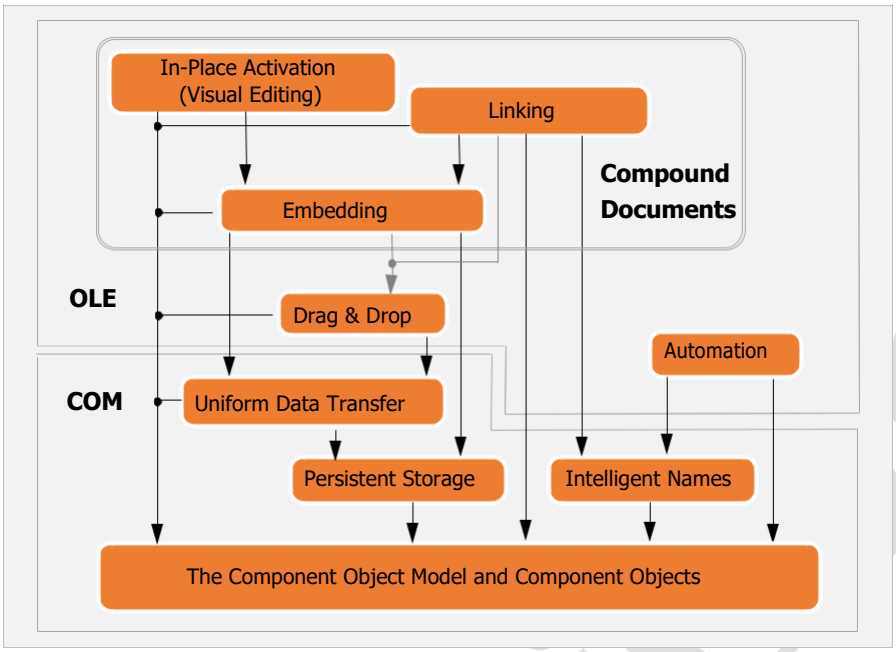


Figure 1.10: OLE builds its features on COM.

1.10 Miscellaneous

1.10.1 Static Linking and Dynamic Linking

In computer science, a library is a collection of non-volatile resources used by computer programs, often to develop software. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications.

Static Linking

A static library or statically-linked library is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable.

Advantages of Static Linking –

The application can be certain that all its libraries are present and that they are the correct version. This avoids dependency problems, known colloquially as DLL Hell or more generally dependency hell. Static linking can also allow the application to be contained in a single executable file, simplifying distribution and installation.
With static linking, it is enough to include those parts of the library that are directly and indirectly referenced by the target executable (or target library). With dynamic libraries, the entire library is loaded, as it is not known in advance which functions will be invoked by applications.

Executes faster than any other linking type

Disadvantages of Static Linking –

In static linking, the size of the executable becomes greater than in dynamic linking, as the library code is stored within the executable rather than in separate files. But if library files are counted as part of the application then the total size will be similar, or even smaller if the compiler eliminates the unused symbols.
When multiple instances of the same program run, a separate instance of the same static library code will exist in the memory per program instance. Thus, memory use is not optimum.
As application code and the library code are strongly coupled, any change in the library code requires rebuilding library code as well as the application code. In short, library cannot be modified independently, client must be rebuilt as well.

Dynamic Linking

A dynamic-link library (DLL) is a module that contains functions and data that can be used by another module (application or DLL) dynamically. A DLL can define two kinds of functions: exported and internal. The exported functions are intended to be called by other modules, as well as from within the DLL where they are defined. Internal functions are typically intended to be called only from within the DLL where they are defined. Although a DLL can export data, its data is generally used only by its functions.

DLLs provide a way to modularize applications so that their functionality can be updated and reused more easily. DLLs also help reduce memory overhead when several applications use the same functionality at the same time, because although each application receives its own copy of the DLL data, the applications share the DLL code.
The Windows application programming interface (API) is implemented as a set of DLLs, so any process that uses the Windows API uses dynamic linking.

Advantages of Dynamic Linking –

Saves memory and reduces swapping. Many processes can use a single DLL simultaneously, sharing a single copy of the DLL in memory. In contrast, Windows must load a copy of the library code into memory for each application that is built with a static link library.
Saves disk space. Many applications can share a single copy of the DLL on disk. In contrast, each application built with a static link library has the library code linked into its executable image as a separate copy.
Upgrades to the DLL are easier. When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
Provides after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was shipped.
Supports multilanguage programs. Programs written in different programming languages can call the same DLL function as long as the programs follow the function's calling convention. The programs and the DLL function must be compatible in the following ways:

- the order in which the function expects its arguments to be pushed onto the stack,
- whether the function or the application is responsible for cleaning up the stack, and
- whether any arguments are passed in registers.

Eases the creation of international versions. By placing resources in a DLL, it is much easier to create international versions of an application. You can place the strings for each language version of your application in a separate resource DLL and have the different language versions load the appropriate resources.

Disadvantages of Dynamic Linking -

The application is not self-contained; it depends on the existence of a separate DLL module. The system terminates processes using load-time dynamic linking if they require a DLL that is not found at process startup and gives an error message to the user. The system does not terminate a process using run-time dynamic linking in this situation, but functions exported by the missing DLL are not available to the program.

Not all programming languages can create DLL, hence language dependent
Loading and unloading of a DLL must be done explicitly using LoadLibrary() and FreeLibrary() APIs respectively. COM internally manages loading and unloading of DLL and developer does not have to worry about it.
LoadLibrary() API which loads the DLL at run time fails if the DLL is not present at the given location. This implies that the DLL must be present at the correct location otherwise client call will not be able to use the DLL.
DLL is not backward compatible. Newer version of a DLL may not be able to serve older clients due to change in the code.
DLL exported functions can be victim of the "name mangling" or "name decoration" mechanism used by different compiler vendors. If this happens, functions with the same name but different arguments cannot be used polymorphic-ally. To avoid such case, use one of following…

1. Use "extern C" keyword to specify C binary standardization. This yields universally standard mangled names.
2. Use module definition file (.def) to export functions using ordinance (private unique number) instead of the function names.

COM uses UNICODE standards to avoid name mangling problem. It uses unique integer identifier to identify a method instead of its name.

1.10.2 Unique Identification Mechanism

As explained earlier, COM uses globally unique identifiers (GUIDs) - 128-bit integers that are virtually guaranteed to be unique in the world across space and time - to identify every interface and every object class and type. These globally unique identifiers are the same as UUIDs (Universally Unique IDs) as defined by DCE. Human-readable

names are assigned only for convenience and are locally scoped. This helps insure that COM components do not accidentally connect to an object or via an interface or method, even in networks with millions of objects.

UUID is unique over space and time. Out of 128-bit, 48-bit are generated using system's network card's address, which is unique all over world. When the system does not have a network card attached, this 48-bit number is created using system timestamp with addition of 12-bit as the timestamp is a 60-bit datatype. This timestamp represents the count of 100 nanoseconds interval. Since 0 hrs: 0 minutes: 0 seconds: 0 milliseconds from 15th Oct 1582; this algorithm will work till year 3400 AD. Means theoretically maximum value generated by this algorithm will be 340,282,366,920,900,000,000,000,000,000,000,000,000.

Developer can use OSF's UuidCreate() API to generate this 128-bit number. COM developers has CoCreateGuid() API – which internally calls UuidCreate() - to generate this number

Syntactically GUID is a C structure as outlined below –

```
typedef struct _GUID
{
   DWORD data1;
   WORD  data2;
   WORD  data3;
   BYTE  data4[8];
};
```

Datatype	Total bytes	Total bits
DWORD	4	32
WORD	2	16
WORD	2	16
BYTE	8	64
	16	128

Bit composition of the GUID type is outlined below –

Microsoft and OSF (Open Source Foundation) together defined GUID type based on the UUID type to identify COM Objects (CoClasses) and Interfaces uniquely. There are more such types defined by Microsoft such as, IID (Interface identifier), CLSID (Class identifier), APPID (Application identifier), DISPID (Dispatch identifier), LCID (Locale identifier), PROGID (Program identifier), etc.

Page intentionally left blank

Page intentionally left blank

2. Component Object Model Technical Overview

Chapter 1 introduced some important challenges and problems in computing today and the Component Object Model as a solution to these problems. Chapter 1 introduced interfaces, mentioned the base interface called IUnknown, and described how interfaces are generally used to communicate between an object and a client of that object, and explained the role that COM has in that communication to provide location transparency.

2.1 Objects and Interfaces

Chapter 1 described that interfaces are - strongly typed semantic contracts between client and object, and that an object in COM is any structure that exposes its functionality through the interface mechanism. In addition, Chapter 1 noted how interfaces follow a binary standard and how such a standard enables clients and objects to interoperate regardless of the programming languages used to implement them. While the *type* of an interface is by colloquial convention referred to with a name starting with an "I" (for interface), this name is only of significance in source-level programming tools. Each interface itself—the immutable contract, that is—as a functional group is referred to at runtime with a globally-unique interface identifier, an "IID" that allows a client to ask an object if it supports the semantics of the interface without unnecessary overhead and without versioning problems. Clients ask questions using a QueryInterface function that all objects support through the base interface, IUnknown.

Furthermore, clients always deal with objects through interface pointers and never directly access the object itself. Therefore an interface is not an object, and an object can, in fact, have more than one interface if it has more than one group of functionality it supports.

Let's now turn to how interfaces manifest themselves and how they work.

2.1.1 Interfaces and C++ Classes

As just reiterated, an interface is not an object, nor is it an object class. Given an interface definition by itself, that is, the type definition for an interface name that begins with "I," you cannot create an object of that type. This is one reason why the prefix "I" is used instead of the common C++ convention of using a "C" to prefix an object class, such as CMyClass. While you can instantiate an object of a C++ class, you cannot instantiate an object of an interface type.

In C++ applications, interfaces are, in fact, defined as *abstract base classes*. That is, the interface is a C++ class that contains nothing but pure virtual member functions. This means that the interface carries no implementation and only prescribes the function signatures for some other class to implement. C++ compilers will generate compile-time errors for code that attempts to instantiate an abstract base class. C++ applications implement COM objects by inheriting these function signatures from one or more interfaces, overriding each interface function, and providing an implementation of each function. This is how a C++ COM application "implements interfaces" on an object.

Implementing objects and interfaces in other languages is similar in nature, depending on the language. In C, for example, an interface is a structure containing a pointer to a table of function pointers, one for each method in the interface. It is very straightforward to use or to implement a COM object in C, or indeed in any programming language which supports the notion of function pointers.

The abstract-base class comparison exposes an attribute of the "contract" concept of interfaces: if you want to implement any single function in an interface, you must provide some implementation for *every* function in that interface. The implementation might be nothing more than a single return statement when the object has nothing to do in that interface function. In most cases there is some meaningful implementation in each function.

A particular object will provide implementations for the functions in every interface that it supports. Objects which have the same set of interfaces and the same implementations for each are often said (loosely) to be instances of the same class because they generally implement those interfaces in a certain way. However, all access to the instances of the class by clients will only be through interfaces; clients know nothing about an object other than it supports certain interfaces. As a result, classes play a much less significant role in COM than they do in other object oriented systems.

COM uses the word "interface" in a sense different from that typically used in object-oriented programming using C++. In the C++ context, "interface" describes *all* the functions that a class supports and that clients of an object can call to interact with it. A COM interface refers to a pre-defined group of related functions that a COM class implements, but does not necessarily represent *all* the functions that the class supports. This separation of an object's

functionality into groups is what enables COM and COM applications to avoid the problems inherent with versioning traditional all-inclusive interfaces.

Client can use OLE/COM Object Viewer (C:\Program Files (x86)\Windows Kits\10\bin\x86\oleview.exe) tool to view component, type library and interfaces related information. In DCOM world, programmer can use IMultiQI interface to query the interfaces supported by a component. It lets you specify an array of structures (MULTI_QI), each containing a pointer to an IID. The purpose of the MULTI_QI structure is to optimize QueryInterface so that fewer round trips are made between machines.

2.1.2 Interface and Inheritance

COM separates class hierarchy (or indeed any other implementation technology) from interface hierarchy and both of those from any implementation hierarchy. Therefore, interface inheritance is only applied to reuse the definition of the contract associated with the base interface. There is no selective inheritance in COM: if one interface inherits from another, it includes all the functions that the other interface defines, for the same reason than an object must implement all interface functions it inherits.

Inheritance is used sparingly in the COM interfaces. Most of the pre-defined interfaces inherit directly from IUnknown (to receive the fundamental functions like QueryInterface), rather than inheriting from another interface to add more functionality. Because COM interfaces are inherited from IUnknown, they tend to be small and distinct from one another. This keeps functionality in separate groups that can be independently updated from the other interfaces, and can be recombined with other interfaces in semantically useful ways.

In addition, interfaces only use single inheritance, never multiple inheritance, to obtain functions from a base interface. Providing otherwise would significantly complicate the interface method call sequence, which is just an indirect function call, and, further, the utility of multiple inheritance is subsumed (contain or include) within the capabilities provided by QueryInterface.

For example: consider a COM interface IStack with 3 functions, Pop, Push and GetCount, inherits from the IUnknwon interface as illustrated below.

```
class IUnknown
{
       HRESULT __stdcall QueryInterface(REFIID, void**);
       HRESULT __stdcall AddRef(void);
       HRESULT __stdcall Release(void);
}
-----------------------------------------------
class IStack : IUnknown
{
   virtual int Pop() = 0;
   virtual void Push(int) = 0;
   virtual int GetCount() = 0;
}
```

Now the order of functions in the VTABLE for this inheritance will be, first comes the IUnknown interface's functions followed by custom interface's (IStack) functions.

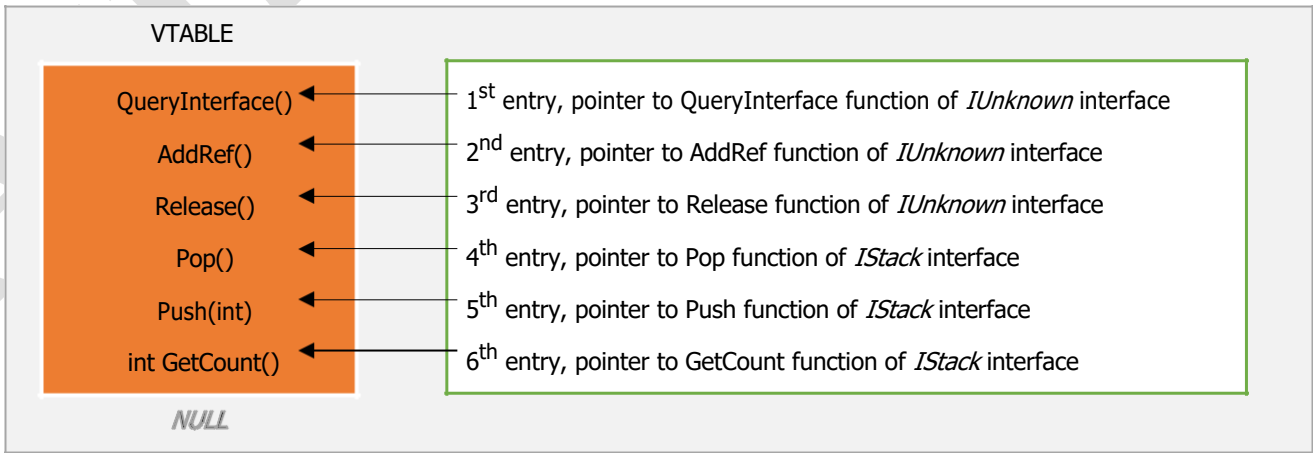


Figure 2.1: VTABLE functions order for the custom COM interface IStack.

2.1.3 Interface Definitions: IDL

When a designer creates an interface, that designer usually defines it using an Interface Description Language (IDL). From this definition an IDL compiler can generate header files for programming languages such that applications can use that interface, create proxy and stub objects to provide for remote procedure calls, and output necessary to enable RPC calls across a network.

IDL is simply a tool for the convenience of the interface designer and is not central to COM's interoperability. It really just saves the designer from manually creating many header files for each programming environment and from creating proxy and stub objects by hand, which would not likely be a fun task.

Later on we will see the Microsoft Interface Description Language (MIDL) compiler in detail. We will also look at Type Libraries which are the machine readable form of IDL, used by tools and other components at runtime.

2.1.4 Basic Operations: IUnknown Interface

All objects in COM, through any interface, allow clients access to two basic operations:

Navigating between multiple interfaces on an object through the QueryInterface function.

Controlling the object's lifetime through a reference counting mechanism handled with functions called AddRef and Release.

Both of these operations as well as the three functions (and only these three) make up the IUnknown interface from which all other interfaces inherit. That is, all interfaces are polymorphic with IUnknown so they all contain QueryInterface, AddRef, and Release functions. As IUnknown interface is at the root of every COM interface, it is also called "root interface".

2.1.4.1 Navigating Multiple Interfaces: the QueryInterface Function

As described in Chapter 1, QueryInterface is the mechanism by which a client, having obtained one interface pointer on a particular object, can request additional pointers to other interfaces on that same object. An input parameter to QueryInterface is the interface identifier (IID) of the interface being requested. If the object supports this interface, it returns that interface on itself through an accompanying output parameter (void **) typed as a generic void; if not, the object returns an error.

In effect, what QueryInterface accomplishes is a switch between contracts on the object. A given interface embodies the interaction that a certain contract requires. Interfaces are groups of functions because contracts in practice invariably require more than one supporting function. QueryInterface separates the request "Do you support a given contract?" from the high-performance use of that contract once negotiations have been successful. Thus, the (minimal) cost of the contract negotiation is reduced over the subsequent use of the contract.

Conversely, QueryInterface provides a robust and reliable way for a component to indicate that in fact does not support a given contract. That is, if using QueryInterface one asks an "old" object whether it supports a "new" interface (one, say, that was invented after the old object has been shipped), then the old object will reliably and robustly answer "no;" the technology which supports this is the algorithm by which IIDs are allocated. While this may seem like a small point, it is excruciatingly important to the overall architecture of the system.

The strengths and benefits of the QueryInterface mechanism need not be reiterated here further, but there is one pressing issue: how does a client obtain its first interface pointer to an object? There are, in fact, four methods through which a client obtains its first interface pointer to a given object:

1. Call a COM Library API function that creates an object of a pre-determined type—that is, the function will only return a pointer to one specific interface for a specific object class.
2. Call a COM Library API function that can create an object based on a class identifier and that returns any type interface pointer requested.
3. Call a member function of some interface that creates another object (or connects to an existing one) and returns an interface pointer on that separate object.
4. Implement an object with an interface through which other objects pass their interface pointer to the client directly. This is the case where the client is an object implementer and passes a pointer to its object to another object to establish a bi-directional connection.

2.1.4.2 Reference Counting: Controlling Object Life-cycle

Just like an application must free memory it allocated once that memory is no longer in use, a client of an object is responsible for freeing the object when that object is no longer needed. In an object-oriented system the client can only do this by giving the object an instruction to free itself.

However, the difficulty lies in having the object know when it is safe to free itself. COM objects, which are dynamically allocated, must allow the client to decide when the object is no longer in use, especially for local or remote objects that may be in use by multiple clients at the same time—the object must wait until *all* clients are finished with it before freeing itself.

COM specifies a *reference counting* mechanism to provide this control. Each object maintains a 32-bit reference count that tracks how many clients are connected to it, that is, how many pointers exist to any of its interfaces in any client. The use of a 32-bit counter (more than four billions clients) means that there's virtually no chance of overloading the count. The two IUnknown functions of AddRef and Release that all objects must implement control the count: AddRef increments the count and Release decrements it. When the reference count is decremented to zero, Release is allowed to free the object because no one else is using it anywhere. Most objects have only one implementation of these functions (along with QueryInterface) that are shared between all interfaces, though this is just a common implementation approach. Architecturally, from a client's perspective, reference counting is strictly and clearly a per-interface notion.

Whenever a client calls a function that returns a new interface pointer to it, such as QueryInterface, the function being called is responsible for incrementing the reference count through the returned pointer. For example, when a client first creates an object it receives back an interface pointer to an object that, from the client's point of view, has a reference count of one. If the client calls QueryInterface once for another interface pointer, the reference count is two. The client must then call Release through *both* pointers (in any order) to decrement the reference count to zero before the object as a whole can free itself. In general, every copy of any pointer to any interface requires a reference count on it.

2.1.5 How an Interface Works

An instantiation of an interface implementation (because the defined interfaces themselves cannot be instantiated without implementation) is simply pointer to an array of pointers to functions. Any code that has access to that array—a pointer through which it can access the array—can call the functions in that interface. In reality, a pointer to an interface is actually a pointer to a pointer to the table of function pointers. This is an inconvenient way to speak about interfaces, so the term "interface pointer" is used instead to refer to this multiple indirection. Conceptually, then, an interface pointer can be viewed simply as a pointer to a function table in which you can call those functions by dereferencing them by means of the interface pointer as shown in Figure 2.2.

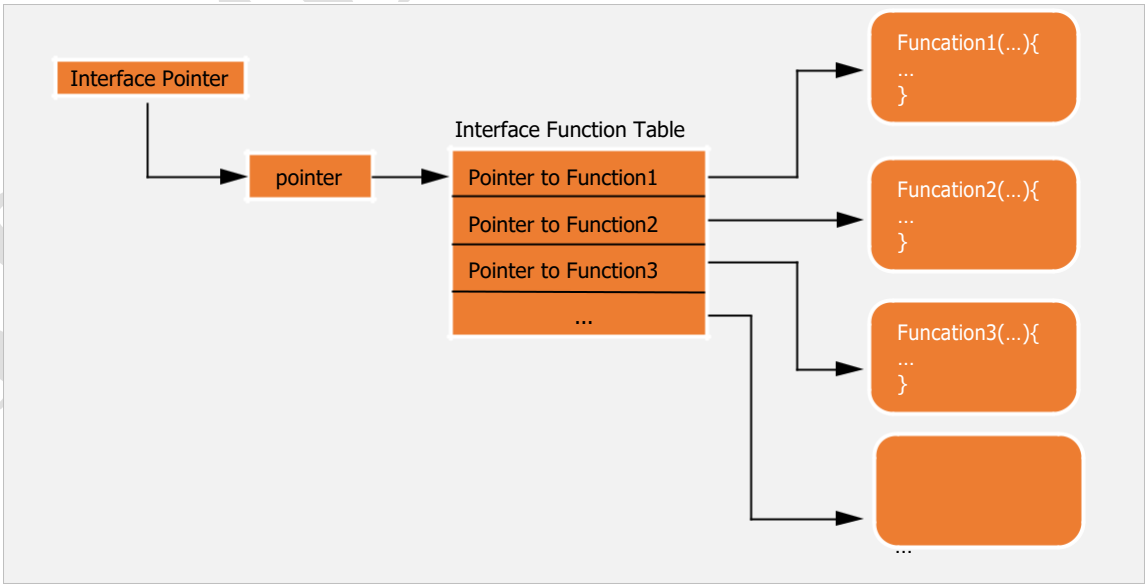
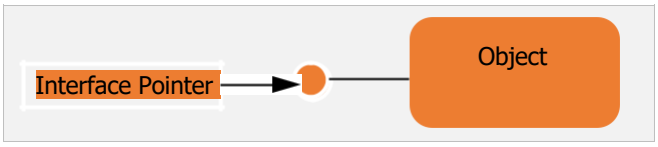


Figure 2.2: An interface pointer is a pointer to a pointer to an array of pointers to the functions in the interface.

Since these function tables are inconvenient to draw they are represented with the "plug-in jack" or "bubbles and push-pins" diagram first shown in Chapter 1 to mean exactly the same thing:



Objects with multiple interfaces are merely capable of providing more than one function table. Function tables can be created manually in a C application or almost automatically with C++ (and other object oriented languages that support COM). With appropriate compiler support (which is inherent in C and C++), a client can call an interface function through the name of the function and not its position in the array. The names of functions and the fact that an interface is a type allows the compiler to check the types of parameters and return values of each interface function call. In contrast, such type-checking is not available even in C or C++ if a client used a position-based calling scheme.

2.1.6 Interfaces Enable Interoperability

COM is designed around the use of interfaces because interfaces enable interoperability. There are three properties of interfaces that provide this: polymorphism, encapsulation, and transparent remoting.

2.1.6.1 Polymorphism

Polymorphism means the ability to assume many forms, and in object-oriented programming it describes the ability to have a single statement invoke different functions at different times. All COM interfaces are polymorphic; when you call a function using an interface pointer, you don't specify which implementation is invoked. A call to pInterface->SomeFunction can cause different code to run depending on what kind of object is the implementer of the interface pointed by pInterface—while the semantics of the function are always the same, the implementation details can vary.

Because the interface standard is a binary standard, clients that know how to use a given interface can interact with any object that supports that interface no matter how the object implements that contract. This allows interoperability as you can write an application that can cooperate with other applications without you knowing who or what they are beforehand.

2.1.6.2 Encapsulation

Other advantages of COM arise from its enforcement of encapsulation. If you have implemented an interface, you can change or update the implementation without affecting any of the clients of your class. Similarly, you are immune to changes that others make in their implementations of their interfaces; if they improve their implementation, you can benefit from it without recompiling your code.

This separation of contract and implementation can also allow you to take advantage of the different implementations underlying an interface, even though the interface remains the same. Different implementations of the same interface are interchangeable, so you can choose from multiple implementations depending on the situation.

Interfaces provides extensibility; a class can support new functionality by implementing additional interfaces without interfering with any of its existing clients. Code using an object's ISomeInterface interface is unaffected if the class is revised to support additional IAnotherInterface interface.

2.1.6.3 Transparent Remoting

COM interfaces allow one application to interact with others anywhere on the network just as if they were on the same machine. This expands the range of an object's interoperability: your application can use any object that supports a given contract, no matter how the object implements that contract, and no matter what machine the object resides on.

Before COM, class code such as C++ class libraries ran in same process, either linked into the executable or as a dynamic-link library. Now class code can run in a separate process, on the same machine or on a different machine, and your application can use it with no special code. COM can intercept calls to interfaces through the function table and generate remote procedure calls instead.

2.2 COM Application Responsibilities

Each process that uses COM in any way—client, server, object implementer—is responsible for three things:

1. Verify that the COM Library is a compatible version with the COM function CoBuildVersion.
2. Initialize the COM Library before using any other functions in it by calling the COM function CoInitialize.
3. Un-initialize the COM Library when it is no longer in use by calling the COM function CoUninitialize.

Note first that most COM Library functions, primarily those that deal with the COM foundation, are prefixed with "Co" to identify their origin. The COM Library may implement other functions to support persistent storage, naming, and data transfer without the "Co" prefix.

OLE library functions start with "OLE" prefix to identify their origin. Though OLE functions can still be used with COM, it is recommended to use its COM equivalent (starts with "Co"prefix) if available.

2.3 Memory Management Rules

In COM there are many interface member functions and APIs which are called by code written by one programming organization and implemented by code written by another. Many of the parameters and return values of these functions are of types that can be passed around by value; however, sometimes there arises the need to pass data structures for which this is not the case, and for which it is therefore necessary that the caller and the callee agree as to the allocation and de-allocation policy. This could in theory be decided and documented on an individual function by function basis, but it is much more reasonable to adopt a universal convention for dealing with these parameters. Also, having a clear convention is important technically in order that the COM remote procedure call implementation can correctly manage memory.

Memory management of pointers to interfaces is always provided by member functions in the interface in question. For all the COM interfaces these are the AddRef and Release functions found in the IUnknown interface, from which again all other COM interfaces derive (as described earlier in this chapter). This section relates only to not-by-value parameters which are *not* pointers to interfaces but are instead more mundane things like strings, pointers to structures, etc.

The COM Library provides an implementation of a memory allocator (see CoGetMalloc and CoTaskMemAlloc). Whenever ownership of an allocated chunk of memory is passed through a COM interface or between a client and the COM library, this allocator must be used to allocate the memory.

Each parameter to and the return value of a function can be classified into one of three groups: an **in** parameter, an **out** parameter (which includes return values), or an **in-out** parameter. In each class of parameter, the responsibility for allocating and freeing non-by-value parameters is the following:

in parameter	Allocated and freed by the caller.
out parameter	Allocated by the callee; freed by the caller.
in-out parameter	Initially allocated by the caller, then freed and re-allocated by the callee if necessary. As with "out" parameters, the caller is responsible for freeing the final returned value.

In the latter two cases there is one piece of code that allocates the memory and a different piece of code that frees it. In order for this to be successful, the two pieces of code must of course have knowledge of which memory allocator is being used. Again, it is often the case that the two pieces of code are written by independent development organizations. To make this work, we require that the COM allocator be used.

Further, the treatment of out and in-out parameters in failure conditions needs special attention. If a function returns a status code which is a failure code, then in general the caller has no way to clean up the *out* or *in-out* parameters. This leads to a few additional rules:

In-out parameter	In error returns, all in-out parameters must either be left alone by the callee (and thus remaining at the value to which it was initialized by the caller; if the caller didn't initialize it, then it's an out parameter, not an in-out parameter) or be explicitly set as in the out parameter error return case.
out parameter	In error returns, out parameters must be always reliably set to a value which will be cleaned up without any action on the caller's part. Further, it is the case that all out pointer parameters

	(usually passed in a pointer-to-pointer parameter, but which can also be passed as a member of a caller-allocate callee-fill structure) must explicitly be set to NULL. The most straightforward way to ensure this is (in part) to set these values to NULL on function entry. (On success returns, the semantics of the function of course determine the legal return values.)

Remember that these memory management conventions for COM applications apply only across public interfaces and APIs—there is no requirement at all that memory allocation strictly internal to a COM application need be done using these mechanisms.

2.4 The COM Client/Service Model

Chapter 1 mentioned how COM supports a model of client/server interaction between a user of an object's services, the client, and the implementer of that object and its services, the server. To be more precise, the client is *any* piece of code (not necessarily an application) that somehow obtains a pointer through which it can access the services of an object and then invokes those services when necessary. The server is some piece of code that implements the object and structures in such a way that the COM Library can match that implementation to a class identifier, or CLSID. The involvement of a class identifier is what differentiates a server from a more general object implementer.

The COM Library uses the CLSID to provide "implementation locator" services to clients. A client only needs to tell COM the CLSID it wants and the type of server—in-process, local, or remote—that it allows COM to load or launch. COM, in turn, locates the implementation of that class and establishes a connection between it and the client. This relationship between client, COM, and server is illustrated in Figure 2.3.

Chapter 1 also introduced the idea of Location transparency, where clients and servers never need to know how far apart they actually are, that is, whether they are in the same process, different processes, or different machines.

This section now takes a closer look at the mechanisms in COM that make this transparency work as well as the responsibilities of client and server applications.

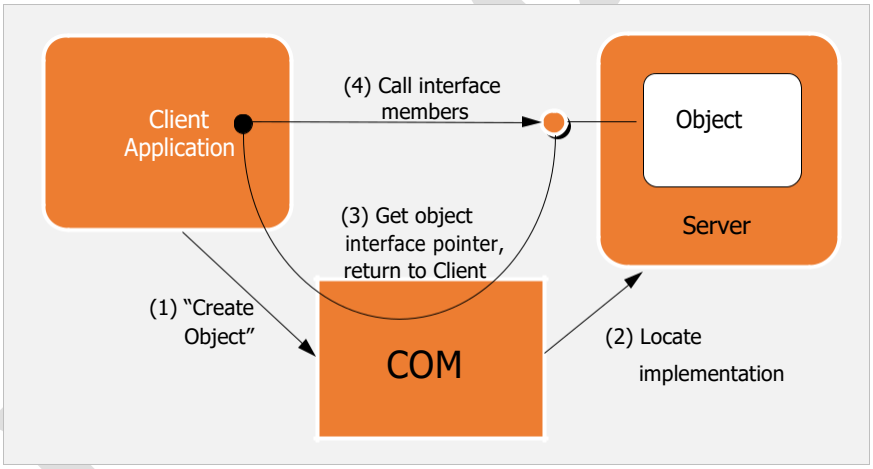


Figure 2.3: Clients locate and access objects through implementation locator services in COM. COM then connects the client to the object in a server. Compare this with Figure 1.1 in Chapter 1.

2.4.1 COM Objects and Class Identifiers

A COM class is a particular implementation of certain interfaces; the implementation consists of machine code that is executed whenever you interact with an instance of the COM class. COM is designed to allow a class to be used by different applications, including applications written without knowledge of that particular class's existence. Therefore class code exists either in a dynamic linked library (DLL) or in another application (EXE). COM specifies a mechanism by which the class code can be used by many different applications.

A COM object is an object that is identified by a unique 128-bit CLSID that associates an object class with a particular DLL or EXE in the file system. A CLSID is a GUID itself (like an interface identifier), so no other class, no matter what vendor writes it, has a duplicate CLSID. Servers implementers generally obtain CLSIDs through the CoCreateGUID function in COM, or through a COM-enabled tool (e.g. C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\Tools\guidgen.exe) that internally calls this function.

The use of unique CLSIDs avoids the possibility of name collisions among classes because CLSIDs are in no way connected to the names used in the underlying implementation. So, for example, two different vendors can write classes which they call "StackClass," but each will have a unique CLSID and therefore avoid any possibility of a collision.

Further, no central authoritative and bureaucratic body is needed to allocate or assign CLSIDs. Thus, server implementers across the world can independently develop and deploy their software without fear of accidental collision with software written by others.

On its host system, COM maintains a registration database (or "registry") of all the CLSIDs for the servers installed on the system, that is, a mapping between each CLSID and the location of the DLL or EXE that houses the server for that CLSID. COM consults this database whenever a client wants to create an instance of a COM class and use its services. That client, however, only needs to know the CLSID which keeps it independent of the specific location of the DLL or EXE on the particular machine.

If a requested CLSID is not found in the local registration database, various other administratively-controlled algorithms are available by which the implementation is attempted to be located on the network to which the local machine may be attached.

Given a CLSID, COM invokes a part of itself called the Service Control Manager (SCM) which is the system element that locates the code for that CLSID. The code may exist as a DLL or EXE on the same machine or on another machine: the SCM isolates most of COM, as well as all applications, from the specific actions necessary to locate code.

2.4.2 COM Clients

Whatever application passes a CLSID to COM and asks for an instantiated object in return is a COM Client. Of course, since this client uses COM, it is also a COM application that must perform the required steps described above and in subsequent chapters.

Regardless of the type of server in use (in-process, local, or remote), a COM Client always asks COM to instantiate objects in exactly the same manner. The simplest method for creating one object is to call the COM function CoCreateInstance. This creates one object of the given CLSID and returns an interface pointer of whatever type the client requests. Alternately, the client can obtain an interface pointer to what is called the "class factory" object for a CLSID by calling CoGetClassObject. This class factory supports an interface called IClassFactory through which the client asks that factory to manufacture an object of its class. At that point the client has interface pointers for two separate objects, the class factory and an object of that class, that each have their own reference counts. It's an important distinction that is illustrated in Figure 2.4.

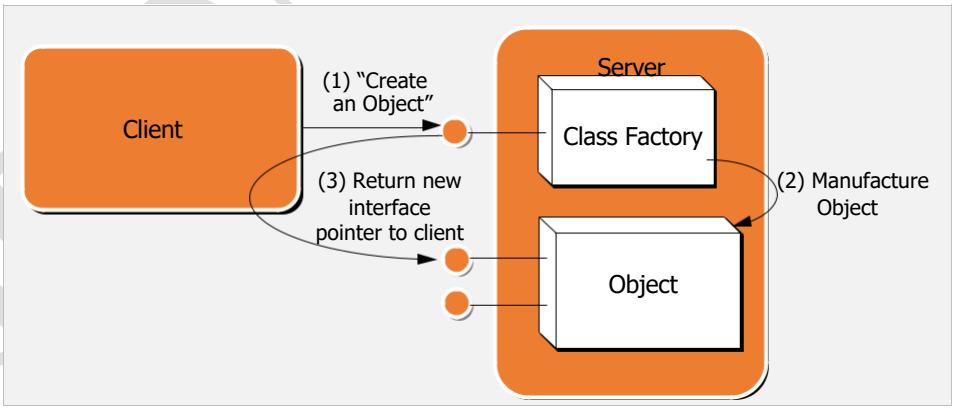


Figure 2.4: A COM Client creates objects through a class factory.

The CoCreateInstance function internally calls CoGetClassObject itself. It's just a more convenient function for clients that want to create one object.

The bottom line is that a COM Client, in addition to its responsibilities as a COM application, is responsible to use COM to obtain a class factory, ask that factory to create an object, initialize the object, and to call that object's (and the class factory's) Release function when the client is finished with it.

2.4.3 COM Servers

There are two basic kinds of object servers:

Dynamic Link Library (DLL) Based: The server is implemented in a module that can be loaded into, and will execute within, a client's address space. (The term DLL is used in this specification to describe any shared library mechanism that is present on a given COM platform.)

EXE Based: The server is implemented as a stand-alone executable module.

Since COM allows for distributed objects, it also allows for the two basic kinds of servers to be implemented on a remote machine. To allow client applications to activate remote objects, COM defines the Service Control Manager (SCM) whose role is described below under "The COM Library."

As a client is responsible for using a class factory and for server management, a server is responsible for implementing the class factory, implementing the class of objects that the factory manufactures, exposing the class factory to COM, and providing for unloading the server under the right conditions. A diagram illustrating what exists inside a server module (EXE or DLL) is shown in Figure 2.5.

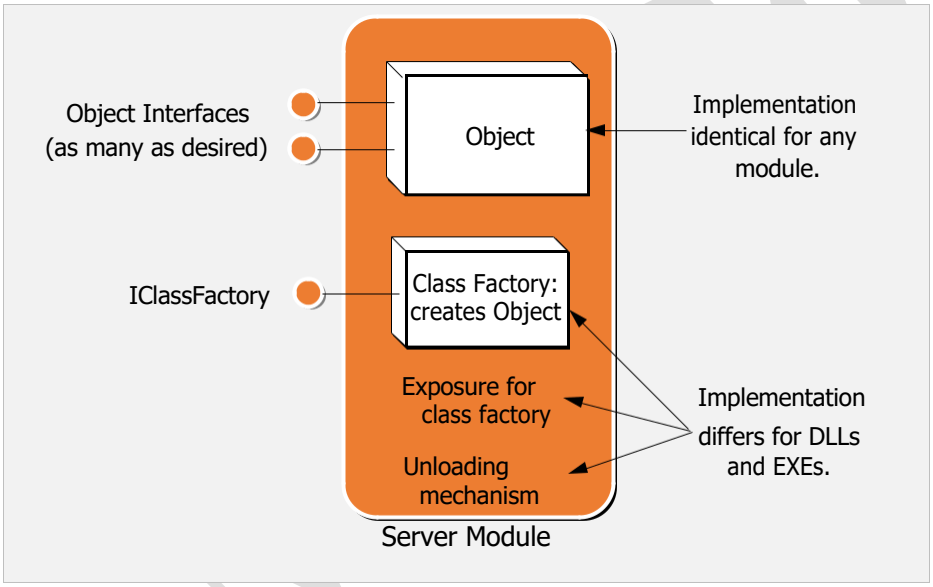


Figure 2.5: The general structure of a COM server.

How a server accomplishes these requirements depends on whether the server is implemented as a DLL or EXE, but is independent of whether the server is on the same machine as the client or on a remote machine. That is, remote servers are the same as local servers but have been registered to be visible to remote clients.

2.5 COM DLL Inproc Server and QI Properties

Now let's look at a sample COM DLL server and a C++ client program that consumes it. Main objective behind this program is to learn more about the QueryInterface method and its principals/rules. We have tested these programs using Visual Studio 2015 community (free) edition.

Program structure:

Server:

1. Header file .h – to declare public interface consumable by clients
2. Source file .cpp – to implement public and private operations

Steps to create the server DLL project:

Step-1: Open Visual Studio 2015 IDE, go to File -> New -> Project option
Step-2: From 'Templates' section, select the Visual C++ option.
Step-3: From 'Visual C++' option, select 'Win32 Project'
Step-4: Follow the 'Win32 Application Wizard' and select following options:

a. Application type: DLL
b. Additional options: Empty project
c. Do not 'Add common header files'

Step-5: Click Finish
Step-6: Add header and source files in the project

Client –

1. Header file – DLL server's header file to get public interfaces
2. Source file – client implementation

Steps to create the client exe project:

Step-1: Open Visual Studio IDE, go to File -> New -> Project option
Step-2: From 'Templates' section, select the Visual C++ option.
Step-3: From 'Visual C++' option, select 'Win32 Project'
Step-4: Follow the 'Win32 Application Wizard' and select following options:
a. Application type: Windows application
b. Additional options: Empty project
c. Do not 'Add common header files'
Step-5: Click Finish
Step-6: Add header and source files in the project

2.5.1 COM DLL Server Implementation

2.5.1.1 Server Header File

Server/component developer is supposed to finalize the functionality first that the component will expose to the client. Once finalized, group that functionality in an interface or multiple interfaces, and declare the component and the interface(s) in the header file. Complete or cut-down version of this header file will be provided to the clients to provide information such as CLSID, interface identifiers, functional capabilities, etc.

QIPropDllServer.h

```
// Contains interfaces, interface identifiers (IIDs), exported functions and other declarations
as // desired.

// Note 1:
// Remember that unlike C++, COM header file only declares the data that is relevant to the
client // and hides the internal implementation related details completely. This syntactical
and binary // encapsulation is one of the critical feature of the COM.

// Note 2:
// COM interface must adhere to these guidelines
// 1. Do not declare any data members in the interface
// 2. Declare all the functions as pure virtual (pure abstract base class)
//    C++ compiler generates a static array called as VTABLE per “interface”/”pure abstract
base // class” and accessed via virtual pointer known as VPTR. At run time, the VTABLE is
brought
//    into the memory and populated with the real/physical address of the virtual functions.
// 3. Declare all the functions as public

// Note 3:
// class and interface (C style struct) both can be used to declare an interface

// Note 4:
// __stdcall keyword denotes the standard/pascal calling convention.
// Unlike C-calling (__cdecl) convention, standard calling convention instructs following -
// 1) Compiler/linker/loader must process the parameters from right to left
// 2) Function in question is responsible to remove the stack memory
// COM follows standard calling convention just like other WIN32 program uses WINAPI, CALLBACK
and // STDMETHODCALLTYPE (nothing but typedef of __stdcall) standard convention
```

```
#pragma once

interface ISum :public IUnknown
{
public:
   // ISum specific method declarations
   virtual HRESULT __stdcall SumOfTwoIntegers(int, int, int *) = 0; // pure virtual
};

class ISubtract :public IUnknown
{
public:
   // ISum specific method declarations
   virtual HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *) = 0; // pure virtual
};


class IMultiply :public IUnknown
{
public:
   // ISum specific method declarations
   virtual HRESULT __stdcall MultiplicationOfTwoIntegers(int, int, int *) = 0;// pure virtual
};

// Note 5:
// IID is –
// 1) 128-bit unique GUID
// 2) used to identify the interface uniquely (interface name is not used for identification)

// IID Of ISum generated using GuidGen.exe tool
const IID IID_ISum = { 0x835970f9,0xa663,0x44b8,0xa9,0x2b,0xa4,0x87,0x53,0x34,0x95,0x63 };

// IID Of ISubtract generated using GuidGen.exe tool
const IID IID_ISubtract = { 0x48d82e54,0x5451,0x40ff,0x80,0x4f,0xe7,0x0,0xa4,0x4a,0x26,0x8a };

// IID Of IMultiply generated using GuidGen.exe tool
const IID IID_IMultiply = { 0xe9df9ebb,0xb9e6,0x4910,0xb2,0x78,0x1b,0x9,0x42,0xc0,0xa2,0x51 };

// exported function declaration

// Note 6:
// extern “C” is used to use C-standard name managing/decoration instead of C++

// Note 7:
// __declspec (dllexport) tells the compiler that the function is exported

// This method creates the CoClass’s instance when called
// In the real world, client calls COM’s CoCreateInstance API to create CoClass instance
extern "C" __declspec (dllexport) HRESULT CreateComponentInstance(REFIID, void **);

// ----- End of the header file
```

2.5.1.2 Server Source CPP File

QIPropDllServer.cpp

```
#define UNICODE

// windows.h header file is required for WIN32 typedefs, MACROS and function prototypes
// It is also required for COM types and functions such as HRESULT, REFIID, IUnknown,
// CoInitialize, etc
#include<windows.h>

#include"QIPropDllServer.h"
```

```
// global variable declarations

class CSumSubtractMultiply :public ISum, ISubtract, IMultiply
{
private:
   ULONG m_cRef;
public:
   // constructor method declarations
   CSumSubtractMultiply(void);

   // destructor method declarations
   ~CSumSubtractMultiply(void);

   // CSumComponent specific methods
   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // ISum specific method declarations (inherited)
   HRESULT __stdcall SumOfTwoIntegers(int, int, int *);

   // ISubtraction specific method declarations (inherited)
   HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *);

   // IMultiplication specific method declarations (inherited)
   HRESULT __stdcall MultiplicationOfTwoIntegers(int, int, int *);
};

// DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
   switch (dwReason)
   {
      case DLL_PROCESS_ATTACH:
         break;
      case DLL_PROCESS_DETACH:
         break;
   }

   return(TRUE);
}

// Implementation Of CSumSubtractMultiply's Constructor Method
CSumSubtractMultiply::CSumSubtractMultiply(void)
{
   m_cRef = 0; // initialization
}

// Implementation Of CSumSubtractMultiply's Destructor Method
CSumSubtractMultiply::~CSumSubtractMultiply(void)
{
   // no code
}

// Implementation Of IUnknown's Methods
HRESULT CSumSubtractMultiply::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<ISum *>(this);
   else if (riid == IID_ISum)
      *ppv = static_cast<ISum *>(this);
   else if (riid == IID_ISubtract)
      *ppv = static_cast<ISubtract *>(this);
   else if (riid == IID_IMultiply)
      *ppv = static_cast<IMultiply *>(this);
   else
   {
```

```
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

   reinterpret_cast<IUnknown *>(*ppv)->AddRef();

   return(S_OK);
}


ULONG CSumSubtractMultiply::AddRef(void)
{
   ++m_cRef;

   return(m_cRef);
}

ULONG CSumSubtractMultiply::Release(void)
{
   --m_cRef;

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of ISum's Methods
HRESULT CSumSubtractMultiply::SumOfTwoIntegers(int num1, int num2, int *pSum)
{
   *pSum = num1 + num2;

   return(S_OK);
}

// Implementation Of ISubtract's Methods
HRESULT CSumSubtractMultiply::SubtractionOfTwoIntegers(int num1, int num2, int *pSubtraction)
{
   *pSubtraction = num1 - num2;

   return(S_OK);
}

// Implementation Of ISum's Methods
HRESULT CSumSubtractMultiply::MultiplicationOfTwoIntegers(int num1, int num2, int
*pMultiplication)
{
   *pMultiplication = num1*num2;

   return(S_OK);
}

// Implementation of exported function CreateComponentInstance()
extern "C" __declspec(dllexport)HRESULT CreateComponentInstance(REFIID riid, void **ppv)
{
   // variable declarations
   CSumSubtractMultiply *pCSumSubtractMultiply = NULL;

   pCSumSubtractMultiply = new CSumSubtractMultiply;
   if (pCSumSubtractMultiply == NULL)
      return(E_OUTOFMEMORY);

   pCSumSubtractMultiply->QueryInterface(riid,
   ppv); return(S_OK);
}
```

2.5.2 COM DLL Client Implementation

2.5.2.1 Client Header File

As discussed earlier, client obtains the header file from the server developer to get CoClass and its interfaces related information. Hence, in our case the header file is same as given in the section 2.5.1.1.

2.5.2.2 Client Source CPP File

ClientOfQIPropDllServer.cpp

```
#define UNICODE
#include<windows.h>
#include"QIPropDllServer.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
void SafeInterfaceRelease(void);

// global variables declarations
ISum *pISum = NULL; ISubtract
*pISubtract = NULL;

// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
   // variable
   declarations WNDCLASSEX
   wndclass; HWND hwnd;
   MSG msg;
   TCHAR AppName[] = TEXT("ComClient");

   // code
   wndclass.cbSize = sizeof(wndclass);
   wndclass.style = CS_HREDRAW | CS_VREDRAW;
   wndclass.cbClsExtra = 0;
   wndclass.cbWndExtra = 0;
   wndclass.lpfnWndProc = WndProc;
   wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
   wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
   wndclass.hInstance = hInstance;
   wndclass.lpszClassName = AppName;
   wndclass.lpszMenuName = NULL;
   wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

   // register window class
   RegisterClassEx(&wndclass);

   // create window
   hwnd = CreateWindow(AppName,
      TEXT("Client Of COM Dll Server"),
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL);


   ShowWindow(hwnd, nCmdShow);
   UpdateWindow(hwnd);
```

```
    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return((int)msg.wParam);
}

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    // function declarations
    BOOL CheckIdentity(ISum *, ISubtract *);
    BOOL CheckPredictability(ISum *);
    BOOL CheckSymmetry(ISum *);
    BOOL CheckReflexivity(ISum *);
    BOOL CheckTransitivity(ISum *);

    // variable declarations
    static HMODULE hServerLib = NULL;
    typedef HRESULT(*PFN_FUNCTION) (REFIID, void **);
    PFN_FUNCTION pfnCreateComponentInstance = NULL;
    HRESULT hr;
    TCHAR str[255];

    // code
    switch (iMsg)
    {
    case WM_CREATE:
        // load the component's dll
                // Make sure that client exe and QIPropDllServer.dll library present in the //
                same directory for LoadLibrary to work
        hServerLib = LoadLibrary(TEXT("QIPropDllServer.dll"));

        if (hServerLib == NULL)
        {
                        MessageBox(hwnd, TEXT("Required Library Can Not Be Loaded"),
                        TEXT("Error"), MB_OK);
            DestroyWindow(hwnd);
        }

        // get pointer to dll's exported "component's instance creating function"
                pfnCreateComponentInstance = (PFN_FUNCTION)GetProcAddress(hServerLib,
                "CreateComponentInstance");
        if (pfnCreateComponentInstance == NULL)
        {
                        MessageBox(hwnd, TEXT("Required Function Can Not Be Obtained"),
                        TEXT("Error"), MB_OK);
            DestroyWindow(hwnd);
        }

        // call "component's instance creating function" to get ISum's
        pointer hr = pfnCreateComponentInstance(IID_ISum, (void **)&pISum);
        if (FAILED(hr))
        {
                        MessageBox(hwnd, TEXT("ISum Interface Can Not Be Obtained."),
                        TEXT("Error"), MB_OK);
            DestroyWindow(hwnd);
        }



        // call ISum's QI() to get ISubtract's pointer
        hr = pISum->QueryInterface(IID_ISubtract, (void **)&pISubtract);
        if (FAILED(hr))
        {
```

```
            MessageBox(hwnd, TEXT("ISubtract Interface Can Not Be Obtained."), TEXT("Error"),
            MB_OK);
            pISum->Release();
            pISum = NULL;
            DestroyWindow(hwnd);
        }

        // check identity
        if (CheckIdentity(pISum, pISubtract) == TRUE)
            wcscpy(str, TEXT("ISum And ISubtract Interfaces Belong To The Same
        Component")); else
            wcscpy(str, TEXT("ISum And ISubtract Interfaces Do Not Belong To The
            Same Component"));
        MessageBox(hwnd, str, TEXT("Message"), MB_OK);

        pISubtract->Release();
        pISubtract = NULL;

        // check predictability
        if (CheckPredictability(pISum) == TRUE)
              wcscpy(str, TEXT("The Component Is Predictable And Its Interfaces Do Not Change
              Over Time"));
        else
              wcscpy(str, TEXT("The Component Is Not Predictable. Its Interfaces Change Over
              Time"));
        MessageBox(hwnd, str, TEXT("Message"), MB_OK);

        // check symmetry
        if (CheckSymmetry(pISum) == TRUE)
            wcscpy(str, TEXT("The Component Is Symmetric"));
        else
            wcscpy(str, TEXT("The Component Is Not Symmetric"));
        MessageBox(hwnd, str, TEXT("Message"), MB_OK);

        // check reflexivity
        if (CheckReflexivity(pISum) == TRUE)
            wcscpy(str, TEXT("The Component Is Reflexive"));
        else
            wcscpy(str, TEXT("The Component Is Not Reflexive"));
        MessageBox(hwnd, str, TEXT("Message"), MB_OK);

        // check transitivity
        if (CheckTransitivity(pISum) == TRUE)
            wcscpy(str, TEXT("The Component Is Transitive"));
        else
            wcscpy(str, TEXT("The Component Is Not Transitive"));
        MessageBox(hwnd, str, TEXT("Message"), MB_OK);

        // exit the application
        DestroyWindow(hwnd);

        break;
    case WM_DESTROY:
        // relinquish the interface
        pointers SafeInterfaceRelease();

        // release the loaded dll
        if (hServerLib)
        {
            FreeLibrary(hServerLib);
            hServerLib = NULL;// make freed module pointer NULL
        }
        PostQuitMessage(0);
        break;
    }

    return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}
```

```
void SafeInterfaceRelease(void)
{
   if (pISum)
   {
      pISum->Release();
      pISum = NULL;
   }

   if (pISubtract)
   {
      pISubtract->Release();
      pISubtract = NULL;
   }
}

BOOL CheckIdentity(ISum *ptrISum, ISubtract *ptrISubtract)
{
   // variable declarations
   IUnknown *pIUnknownFromISum = NULL;
   IUnknown *pIUnknownFromISubtract = NULL;
   HRESULT hr;

   hr = ptrISum->QueryInterface(IID_IUnknown, (void
   **)&pIUnknownFromISum); if (FAILED(hr))
      MessageBox(NULL, TEXT("Can Not Obtain Iunknown From ISum"), TEXT("CheckIdentity()
      Error"), MB_OK);

   hr = ptrISubtract->QueryInterface(IID_IUnknown, (void
   **)&pIUnknownFromISubtract); if (FAILED(hr))
      MessageBox(NULL, TEXT("Can Not Obtain Iunknown From ISubtract"),
      TEXT("CheckIdentity() Error"), MB_OK);

   if (pIUnknownFromISum == pIUnknownFromISubtract)
      return(TRUE);

   return(FALSE);
}

BOOL CheckPredictability(ISum *ptrISum)
{
   // variable declarations
   ISubtract *pISubtract =
   NULL; HRESULT hr;

   hr = ptrISum->QueryInterface(IID_ISubtract, (void
   **)&pISubtract); if (FAILED(hr))
        MessageBox(NULL, TEXT("Can Not Get ISubtract Interface From ISum
        Interface"), TEXT("CheckPredictability() Error"), MB_OK);
   else
   {
      // deliberately release newly obtained ISubtract interface
      pISubtract->Release();
      pISubtract = NULL;

      // wait for some time
      Sleep(1000);// 1 second

      // again obtain ISubtract interface
      hr = ptrISum->QueryInterface(IID_ISubtract, (void
      **)&pISubtract); if (FAILED(hr))
           MessageBox(NULL, TEXT("Can Not Get ISubtract Interface From ISum
           Interface"), TEXT("CheckPredictability() Error"), MB_OK);
      else
      {
         // this is usual release of ISubtract interface
         pISubtract->Release();
         pISubtract = NULL;
```

```
            return(TRUE);
        }
    }

    return(FALSE);
}

BOOL CheckSymmetry(ISum *ptrISum)
{
    // variable declarations
    ISubtract *pISubtract =
    NULL; ISum *pISumAgain =
    NULL; HRESULT hr;

    // code
    hr = ptrISum->QueryInterface(IID_ISubtract, (void
    **)&pISubtract); if (FAILED(hr))
        MessageBox(NULL, TEXT("Can Not Get ISubtract Interface From ISum
        Interface"), TEXT("CheckSymmetry() Error"), MB_OK);
    else
    {
        // again obtain ISum interface
        hr = pISubtract->QueryInterface(IID_ISum, (void **)&pISumAgain);
        if (FAILED(hr))
            MessageBox(NULL, TEXT("Can Not Get ISum Interface From ISubtract
            Interface"), TEXT("CheckSymmetry() Error"), MB_OK);
        else
        {
            if (ptrISum == pISumAgain)
            {
                pISumAgain->Release();
                pISumAgain = NULL;
                return(TRUE);
            }
            else
            {
                pISumAgain->Release();
                pISumAgain = NULL;
                return(FALSE);
            }
        }
    }

    return(FALSE);
}

BOOL CheckReflexivity(ISum *ptrISum)
{
    // variable  declarations
    ISum *pISumAgain = NULL;
    HRESULT hr;

    // code
    hr = ptrISum->QueryInterface(IID_ISum, (void
    **)&pISumAgain); if (FAILED(hr))
        MessageBox(NULL, TEXT("Can Not Get ISum Interface From ISum Interface Itself"),
        TEXT("CheckReflexivity() Error"), MB_OK);
    else
    {
        if (ptrISum == pISumAgain)
        {
            pISumAgain->Release();
            pISumAgain = NULL;
            return(TRUE);
        }
        else
        {
            pISumAgain->Release();
```

```
            pISumAgain = NULL;
            return(FALSE);
        }
    }

    return(FALSE);
}

BOOL CheckTransitivity(ISum *ptrISum)
{
    // variable      declarations
    ISum   *pISumAgain   =   NULL;
    ISubtract    *pISubtract    =
    NULL;  IMultiply  *pIMultiply
    = NULL; HRESULT hr;

    // code
    // get ISubtract interface
    hr = ptrISum->QueryInterface(IID_ISubtract, (void
    **)&pISubtract); if (FAILED(hr))
        MessageBox(NULL, TEXT("Can Not Get ISubtract Interface From ISum
        Interface"), TEXT("CheckTransitivity() Error"), MB_OK);

    // get IMultiply interface
    hr = pISubtract->QueryInterface(IID_IMultiply, (void
    **)&pIMultiply); if (FAILED(hr))
        MessageBox(NULL, TEXT("Can Not Get IMultiply Interface From ISubtract Interface"),
        TEXT("CheckTransitivity() Error"), MB_OK);

    // again get ISum interface
    hr = pIMultiply->QueryInterface(IID_ISum, (void
    **)&pISumAgain); if (FAILED(hr))
        MessageBox(NULL, TEXT("Can Not Get ISum Interface From IMultiply
        Interface"), TEXT("CheckTransitivity() Error"), MB_OK);
    else
    {
        if (ptrISum == pISumAgain)
        {
            pISumAgain->Release();
            pISumAgain = NULL;
            pIMultiply->Release();
            pIMultiply = NULL;
            pISubtract->Release();
            pISubtract = NULL;
            return(TRUE);
        }
        else
        {
            pISumAgain->Release();
            pISumAgain = NULL;
            pIMultiply->Release();
            pIMultiply = NULL;
            pISubtract->Release();
            pISubtract = NULL;
            return(FALSE);
        }
    }

    return(FALSE);
}
```

You can put a breakpoint at the CheckIdentity call in the client program and in the Autos window can see the corresponding VTABLE structure for ISum and ISubtract interfaces. Make sure that symbols (.pdb) are loaded for the server application (dll). A sample view is given below for your reference:

Autos

Name	Value
CSumSubtractMultiply::QueryIr	S_OK
&plSubtract	0x0018f310 {0x00899cf4 {...}}
	0x00899cf4 {m_cRef=2 }
[CSumSubtractMultiply]	{m_cRef=2 }
IUnknown	{...}
__vfptr	0x5e6f78c0 {QIPropServer.dll!const CSumSubtractMultiply::`vftable'{for `ISubtract'}} {0x5e6f132a {QIPropServer.dll![thunk]:
[0]	0x5e6f132a {QIPropServer.dll![thunk]:CSumSubtractMultiply::QueryInterface`adjustor{4}' (struct _GUID const &,void * *)}
[1]	0x5e6f1195 {QIPropServer.dll![thunk]:CSumSubtractMultiply::AddRef`adjustor{4}' (void)}
[2]	0x5e6f1357 {QIPropServer.dll![thunk]:CSumSubtractMultiply::Release`adjustor{4}' (void)}
IID_ISubtract	{48D82E54-5451-40FF-804F-E700A44A268A}
hr	S_OK
plSum	0x00899cf0 {m_cRef=2 }
[CSumSubtractMultiply]	{m_cRef=2 }
IUnknown	{...}
__vfptr	0x5e6f78a8 {QIPropServer.dll!const CSumSubtractMultiply::`vftable'{for `ISum'}} {0x5e6f101e {QIPropServer.dll!CSumSubtra
[0]	0x5e6f101e {QIPropServer.dll!CSumSubtractMultiply::QueryInterface(struct _GUID const &,void * *)}
[1]	0x5e6f1271 {QIPropServer.dll!CSumSubtractMultiply::AddRef(void)}
[2]	0x5e6f13c0 {QIPropServer.dll!CSumSubtractMultiply::Release(void)}

2.6 The Rules of the Component Object Model

As stated earlier, QueryInterface is the mechanism by which a client, having obtained one interface pointer on a particular object, can request additional pointers to other interfaces on that same object. QueryInterface has ability to switch between contracts on the object and offers a robust and reliable way for a component to indicate that in fact does not support a given contract. This ability makes QueryInterface the most important piece of every COM component. An object is not a Microsoft Component Object Model (COM) object unless it implements at least one interface that at minimum is IUnknown.

2.6.1 Interface Design Rules

Interfaces must directly or indirectly inherit from IUnknown.

Interfaces must have a unique interface identifier (IID).

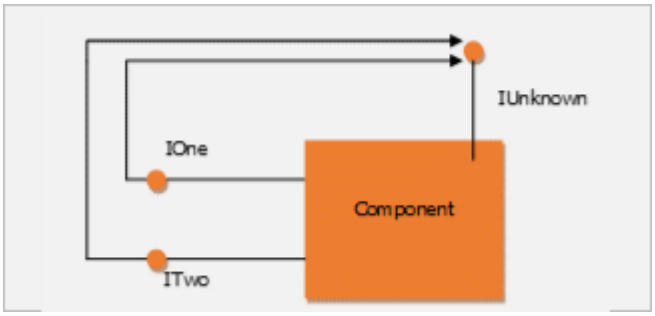
Interfaces are immutable. Once assigned an IID and published, no element of the interface definition may change.

Interface member functions should have a return type of HRESULT to allow the remoting infrastructure to report remote procedure call (RPC) errors.

String parameters in interface member functions should be Unicode.

2.6.2 Implementing IUnknown

1. **Identity**: It is required that every object instance has only one IUknown interface. So call to QueryInterface on any interface for a given object instance for the specific interface IUnknown must always return the same physical pointer value. Client side **`CheckIdentity`** function in the earlier section verifies this rule. **`CheckIdentity`** calls `QueryInterface(IID_IUnknown, ...)` on two interfaces and compares the result to determine whether they point to the same instance of an object. This behaviour can help to improve the performance at run time as only IUnknown must return exactly the same pointer value (hence static) and other interfaces (Tear-off/transient interfaces) can be loaded dynamically on demand saving memory required for vptrs, especially when a component implements a large number of interfaces and all the interfaces are used at all times.



2. **Predictability**: It is required that the set of interfaces accessible on an object via QueryInterface be static over time, not dynamic. That is, if QueryInterface succeeds for a given IID once, it will always succeed on subsequent calls on the same object (except in catastrophic failure situations), and if QueryInterface fails for a given IID, subsequent calls for the same IID on the same object must also fail. Client side `CheckPredictabilityfuncation` function in the earlier section verifies this rule.

3. **Symmetric**, **Reflexive**, and **Transitive** with respect to the set of interfaces that are accessible. Client side `CheckSymmetry`, `CheckReflexivity`, `CheckTransitivity` methods in the earlier sections verify these rules. These rules are explained below.

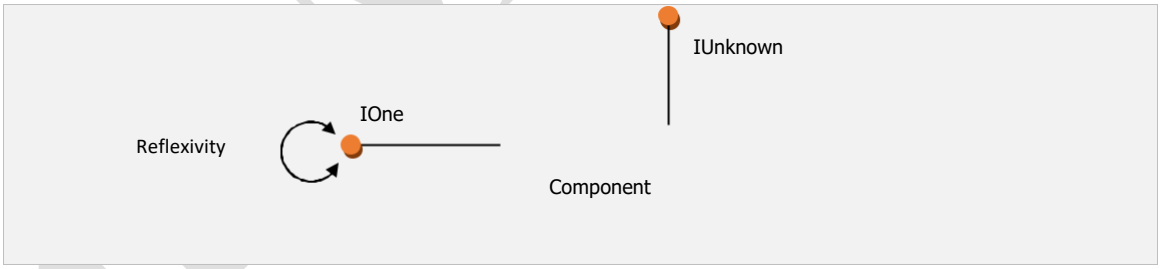
```
Line 1: ISum * pSum = (some function returning ISum pointer);
Line 2: ISubtract * pSubtract = NULL;
Line 3: HRESULT hr;
Line 4: hr = pSum->QueryInterface(IID_ISubtract, &pSubtract);
```

Symmetric: You can always get the interface you have.


```
ISum * pSumAgain = NULL;
pSum->QueryInterface(IID_ISum,&pSumAgain);
```

must succeed (a>>a).

Reflexive: You can always get back to where you started.

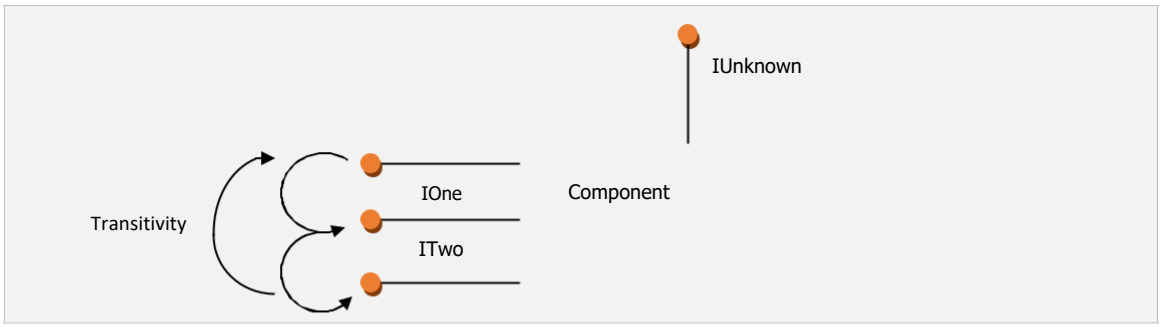


If, in line 4, pSubtract was successfully obtained, then

```
ISum * pSumAgain = NULL;
pSubtract->QueryInterface(IID_ISum, &pSumAgain);
```

must succeed (a>>b, then b>>a).

Transitive: You can get there from anywhere if you can get there from somewhere.



If, in line 4, pSubtract was successfully obtained, and we do

```
IMultiply * pMultiply = NULL;
pSubtract->QueryInterface(IID_IUmltiply, &pMultiply); // line 6
```

and pMultiply is successfully obtained in line 6, then

```
IMultiply * pMultiplyAgain = NULL;
pSum->QueryInterface(IID_IMultiply, &pMultiplyAgain);
```

must succeed (a>>b, and b>>c, then a>>c).

2.6.3 Immutable Interfaces

Chapter 1 described that interfaces are immutable and never versioned, thus avoiding versioning problems. A new version of an interface, created by adding or removing functions or changing semantics, is an entirely new interface and is assigned a new unique identifier. Therefore a new interface does not conflict with an old interface even if all that changed is the semantics. Follow below guidelines when to create a new interface with new unique IID.

One or more functions need to be added
One or more exiting functions need to be removed
Order of existing functions need to be changed
Number or order of parameters of one or more functions need to be changed
Any change in function definition such as parameters, return type needed
[In, Out] memory management parameters need to be
changed Function implementation needs to be changed

2.6.4 HRESULT

Every COM interface function ideally should return HRESULT type that indicates the result of the function behaviour. In the case of QueryInterface, S_OK is returned if the requested interface is found, E_NOINTERFACE otherwise.

Note that HRESULT though start with "H", considering WIN32 convention, it should be a handle, but it is not. It is just a plain 32-bit int type and its values are defined in the winerror.h header file (windows.h internally refers it).

An HRESULT is COM's key error reporting type. In addition, the COM Library provides a few functions and macros to help applications of any kind deal with error information. An HRESULT is a simple 32-bit value:

```
typedef LONG HRESULT;
```

An HRESULT is divided up into an internal structure that has four fields with the following format (numbers indicate bit positions):

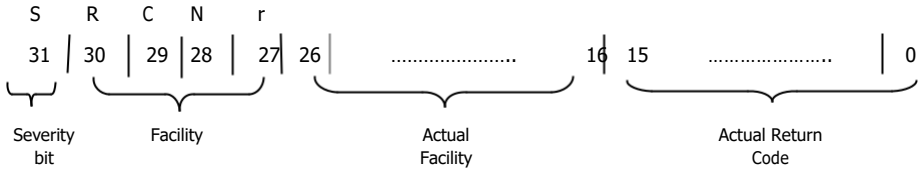


Figure 2.6: HRESULT Structure

S: (1 bit) Severity field:
0 *Success*. The function was successful; it behaved according to its proscribed semantics.
1 *Error*. The function failed due to an error condition.

R: (2 bits) Reserved for future use; must be set to zero by present programs generating HRESULTs; present code should not take action that relies on any particular bits being set or cleared this field.

Facility: (13 bits) Indicates which group of status codes this belongs to. New facilities must be allocated by a central coordinating body since they need to be universally unique. However, the need for new facility codes is very small. Most cases can and should use FACILITY_ITF.

Code: (16 bits) Describes what actually took place, error or otherwise.

Return code:

First 16-bit (0-15) represents the actual return code. The general "success" HRESULT is named S_OK, meaning "everything worked" as per the function specification. The value of this HRESULT is zero. In addition, as it is useful to have functions that can succeed but return Boolean results, the code S_FALSE is defined are success codes intended to mean "function worked and the result is false."

```
#define S_OK       0
#define S_FALSE    1
```

Return codes that start with "S" usually indicate success and return codes that start with "E" usually indicate error. Following are some common return codes.

S_OK: success (similar to Boolean true)
NOERROR: same as S_OK
S_FALSE: success (similar to Boolean false)
E_OUTOFMEMORY: Error (failed to allocate desired memory)
E_FAIL: Error (unspecified)
E_UNEXPECTED: Error (unexpected failure)
E_NOTIMPL: Error (Not implemented)
E_NOINTERFACE: Error (Component does not support the requested interface)
E_INVALIDARG: Error (Invalid argument)
E_POINTER: Error (Invalid pointer passed in to a function)
E_HANDLE: Error (Invalid handle passed in to a function)
E_ABORT: Error (operation aborted)
E_ACCESSDENIED: Error (access denied)

Some important points about the HRESULT:

1. GetLastError() API cannot be used to retrieve the calling thread's last-error code (HRESULT) value reported by a COM API.
2. Use FormatMessage() WIN32 API to get extended error information associated with an HRESULT.
3. You can also use C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\Tools\errlook.exe MFC application to get the extended error information associated with an HRESULT.
4. Use MAKE_HRESULT macro to create an **HRESULT** value from its component pieces.

```
HRESULT MAKE_HRESULT(
   WORD sev, // The severity.
   WORD fac, // The facility.
   WORD code // The code
);
```

Facility code:

Bits from 16 to 30 are called facility code which is reserved for Operating system purpose only. Out of 15 bits, first 11 bits i.e. 16 to 26 are the actual facility code. While the remaining four bits are r bit, N bit, C bit and R bit respectively. COM presently defines the following facility codes:

Facility Name	Facility Value	Description
FACILITY_NULL	0	Used for broadly applicable common status codes that have no specific grouping. S_OK belongs to this facility, for example.
FACILITY_ITF	4	Used for by far the majority of result codes that are returned from an interface member function. Use of this facility indicates that the meaning of the error code is defined solely by the definition of the particular interface in question; an HRESULT with exactly the same 32-bit value returned from another interface might have a different meaning
FACILITY_RPC	1	Used for errors that result from an underlying remote procedure call implementation. In general, this specification does not explicitly document the RPC errors that can be returned from functions, though they nevertheless can be returned in situations where the interface being used is in fact remoted
FACILITY_DISPATCH	2	Used for IDispatch-interface-related status codes.
FACILITY_STORAGE	3	Used for persistent-storage-related status codes. Status codes whose code (lower 16 bits) value is in the range of DOS error codes (less than 256) have the same meaning as the corresponding DOS error.
FACILITY_WIN32	7	Used to provide a means of mapping an error code from a function in the Win32 API into an HRESULT. The semantically significant part of a Win32 error is 16 bits large.
FACILITY_WINDOWS	8	Used for additional error codes from Microsoft-defined interfaces.
FACILITY_CONTROL	10	Used for OLE Controls-related error values.

Severity code: the severity bit (31st bit) has only two values SEVERITY_SUCCESS (0) and SEVERITY_ERROR (1).

To fetch error code, facility code and severity of an HRESULT, use `HRESULT_CODE(HRESULT hr)`, `HRESULT_FACILITY(HRESULT hr)`, and `HRESULT_SEVERITY(HRESULT hr)` MACROS respectively.

2.6.7 Cast Operations

A cast is a special operator that forces one data type to be converted into another. As an operator, a cast is unary and has the same precedence as any other unary operator. The most general cast supported by most of the C++ compilers is as follows:

```
(type) expression
```

Where type is the desired data type. There are other casting operators supported by C++, they are listed below:

const_cast<type> (expr): The const_cast operator is used to explicitly override const and/or volatile in a cast. The target type must be the same as the source type except for the alteration of its const or volatile attributes. This type of casting manipulates the const attribute of the passed object, either to be set or removed.

dynamic_cast<type> (expr): The dynamic_cast performs a runtime cast that verifies the validity of the cast. If the cast cannot be made, the cast fails and the expression evaluates to null. A dynamic_cast performs casts on polymorphic types and can cast a A* pointer into a B* pointer only if the object being pointed to actually is a B object.

reinterpret_cast<new_type> (expr): Converts between types by reinterpreting the underlying bit pattern. The reinterpret_cast operator changes a pointer to any other type of pointer. It also allows casting from pointer to an integer type and vice versa. Unlike static_cast, but like const_cast,

the reinterpret_cast expression does not compile to any CPU instructions. It is purely a compiler directive which instructs the compiler to treat the sequence of bits (object representation) of *expr* as if it had the type *new_type*.

static_cast<type> (expr): The static_cast operator performs a non-polymorphic cast. For example, it can be used to cast a base class pointer into a derived class pointer.

2.6.7.1 Static Casting of Interfaces

Observe the implementation of the `IUnknown->QueryInterface` function in our last program. The implementation somewhat simulates the effects of RTTI (Run Time Type Identification) by navigating the type hierarchy of the CoClass object. Figure 2.6 illustrates the type hierarchy of the `CSumSubtractMultiply` class shown in our last server example. Because the implementation class derives from each interface that it exposes, `CSumSubtractMultiply`'s `QueryInterface` can use explicit static casts to limit the scope of the "this" pointer based on the subtype requested by the client.

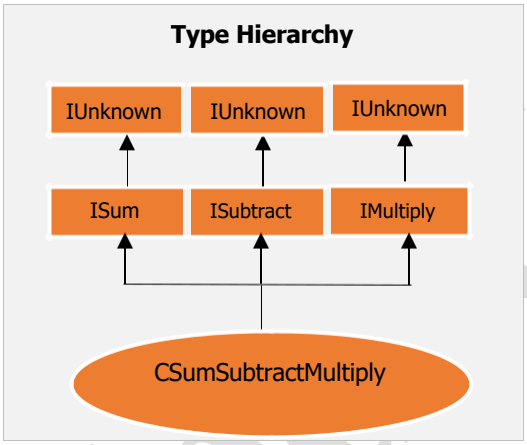


Figure 2.7 Type hierarchy of CSumSubtractMultiply

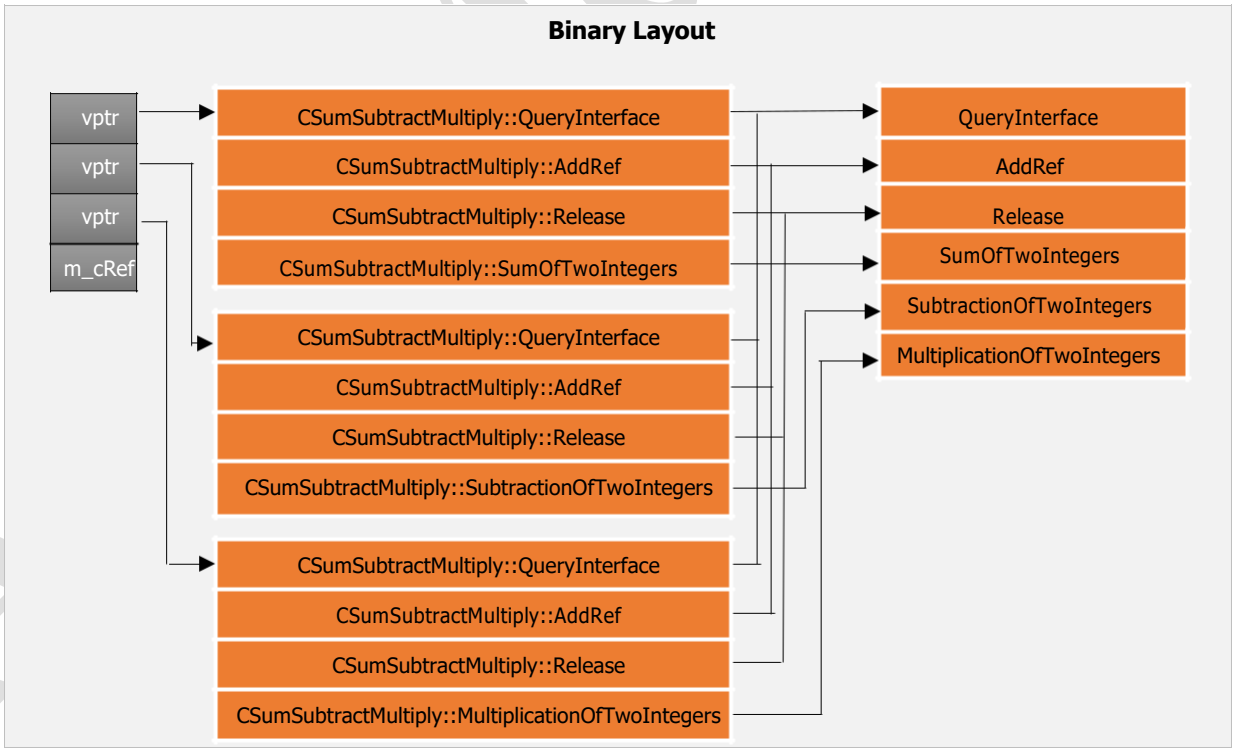


Figure 2.8 Binary Format of CSumSubtractMultiply

Because the object derives from the type (ISum, ISubtract, & IMultiply) used in the cast, the compiled versions of the cast statements simply add a fixed offset to the object's "this" pointer to find the beginning of the base class's layout.

Note that when asked for the common base interface IUnknown, the implementation statically casts itself to ISum. This is because `*ppv = static_cast<IUnknown *>(this);` would be ambiguous as all 3 interfaces ISum, ISubtract and IMultiply derive from IUnknown.

2.6.7.2 Reinterpret casting

In general, for simple types (non-multiple inheritance) reinterpret_cast<>() and static_cast<>() are indeed semantically the same. However, for complex types the difference is that:

`reinterpret_cast<IUnknown *>(*ppv)->AddRef();` is the same as:

```
((IUnknown *)(void*)(*ppv)->AddRef())->AddRef();
```

which throws away any compiler known type information.

static_cast<>(), however, may need to offset the pointer when doing the cast because of the difference in vtbl and local variable offsets between the various types.

In case of COM, QueryInterface must give a valid IUnknown pointer at compile time. Rest of the custom interfaces can wait till runtime. So in general cases where there is a relationship/compatibility between the types `reinterpret_cast<IUnknown *>(*ppv)->AddRef()` is not required, `static_cast<IUnknown *>(*ppv)->AddRef()` can work.

2.7 Adjustor Thunk

In the computer world, a thunk function is used to implement C++ virtual function calls with multiple inheritance. The thunk acts as a wrapper around a virtual function, adjusting the implicit object parameter before handing control off to the real function. To understand adjustor thunk, let's dig more into the `CSumSubtract` COM class and find out how QueryInterface, AddRef and Release methods of ISum and ISubtract are dereferenced internally at runtime by Microsoft C++ compiler.

```
class CSumSubtract :public ISum, ISubtract
{
private:
   ULONG m_cRef;
public:
   CSumSubtractMultiply(void);
   ~CSumSubtractMultiply(void);

       HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   HRESULT __stdcall SumOfTwoIntegers(int, int, int *);
   HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *);
};
```

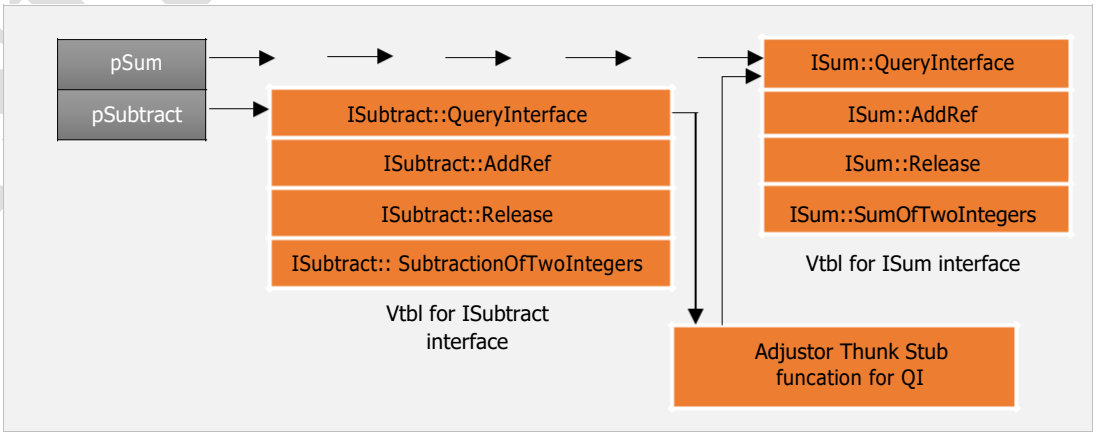


Figure 2.9 Windows uses Adjustor thunk function to implement C++ virtual function calls with multiple inheritance

In the diagram, pSum is the pointer returned when the ISum interface is needed, and pSubtract is the pointer for the ISubtract interface. Now, there is only one QueryInterface method, but there are two entries, one for each vtable. Remember that each function in a vtable receives the corresponding interface pointer as its "this" parameter. That's just fine for ISum::QueryInterface; its interface pointer is the same as the object's interface pointer. But that's bad news for ISubtract::QueryInterface, since its interface pointer is pSubtract, not pSum.

This is where the adjustor thunks come in.

The entry for ISubtract::QueryInterface is a stub function that changes pSubtract to pSum, and then lets ISum::QueryInterface do the rest of the work. This stub function is the adjustor thunk.

```
[thunk]:CSumSubtract::QueryInterface`adjustor{4}':
  sub     DWORD PTR [esp+4], 4 ; this -= sizeof(lpVtbl)
  jmp     CSumSubtract::QueryInterface
```

The adjustor thunk takes the "this" pointer and subtracts 4, converting q into p, then it jumps to the QueryInterface (1) function to do the real work.

Whenever you have multiple inheritance and a virtual function is implemented on multiple base classes, you will get an adjustor thunk for the second and subsequent base class methods in order to convert the "this" pointer into a common format. The entries in the vtble for the leftmost base class (ISum in our case) do not require an adjustor thunk to adjust the "this" pointer prior to entering the method implementation. This applies only to compilers that use adjustor thunks and place the leftmost base at the top of the object layout. Microsoft C++ compiler works this way.

2.8 Reference Counting

Objects accessed through interfaces use a reference counting mechanism to ensure that the lifetime of the object includes the lifetime of references to it. This mechanism is adopted so that independent components can obtain and release access to a single object, and not have to coordinate with each other over the lifetime management. Client is not burdened with the reference counting; instead server does the lifetime management so that the server is available to the client in ready-to-use form. In a sense, the object provides this management, so long as the client components conform to the rules. Within a single component that is completely under the control of a single development organization, clearly that organization can adopt whatever strategy it chooses. The following rules are about how to manage and communicate interface instances between components, and are a reasonable starting point for a policy within a component.

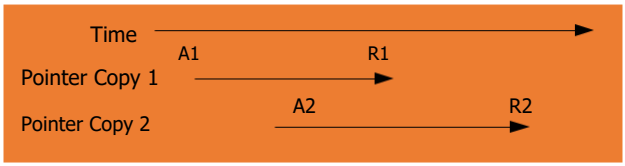
Note that the reference counting paradigm applies only to pointers to interfaces; pointers to data are not referenced counted. Though clients do not have to bother with the component memory management tasks, client must ensure that interface pointers are relinquished timely as explained below.

It is important to be very clear on exactly when it is necessary to call AddRef and Release through an interface pointer. By its nature, pointer management is a cooperative effort between separate pieces of code, which must all therefore cooperate in order that the overall management of the pointer be correct. The conceptual model is the following: interface pointers are thought of as living in pointer variables, which for the present discussion will include variables in memory locations and in internal processor registers, and will include both programmer- and compiler-generated variables. In short, it includes all internal computation state that holds an interface pointer. Assignment to or initialization of a pointer variable involves creating a *new copy* of an already existing pointer: where there was one copy of the pointer in some variable (the value used in the assignment/initialization), there is now two. An assignment to a pointer variable *destroys* the pointer copy presently in the variable, as does the destruction of the variable itself (that is, the scope in which the variable is found, such as the stack frame, is destroyed).

Rule 1: AddRef must be called for every new copy of an interface pointer, and Release called every destruction of an interface pointer except where subsequent rules explicitly permit otherwise.

This is the default case. In short, unless special knowledge permits otherwise, the worst case must be assumed. The exceptions to Rule 1 all involve knowledge of the relationships of the lifetimes of two or more copies of an interface pointer. In general, they fall into two categories.

Category 1. Nested lifetimes



Category 2. Staggered overlapping lifetimes

In Category 1 situations, the AddRef A2 and the Release R2 can be omitted, while in Category 2, A2 and R1 can be eliminated.

Rule 2: Special knowledge on the part of a piece of code of the relationships of the beginnings and the endings of the lifetimes of two or more copies of an interface pointer can allow AddRef/Release pairs to be omitted.

The following rules call out specific common cases of Rule 2. The first two of these rules are particularly important, as they are especially common.

Rule 2a: *In-parameters to functions.* The copy of an interface pointer which is passed as an actual parameter to a function has a lifetime which is nested in that of the pointer used to initialize the value. The actual parameter therefore need not be separately reference counted.

Rule 2b: *Out-parameters from functions, including return values.* This is a Category 2 situation. In order to set the out parameter, the function itself by Rule 1 must have a stable copy of the interface pointer. On exit, the responsibility for releasing the pointer is transferred from the callee to the caller. The out-parameter thus need not be separately reference counted. For example, QueryInterface behaviour, it calls the "AddRef" method before returning a valid interface pointer and calling "Release" is taken care by its caller.

Rule 2c: *Local variables.* A function implementation clearly has omniscient (wise) knowledge of the lifetimes of each of the pointer variables allocated on the stack frame. It can therefore use this knowledge to omit redundant AddRef/Release pairs.

Rule 2d: *Back-pointers*. Some data structures are of the nature of containing two components, A and B, each with a pointer to the other. If the lifetime of one component (A) is known to contain the lifetime of the other (B), then the pointer from the second component back to the first (from B to A) need not be reference counted. Often, avoiding the cycle that would otherwise be created is important in maintaining the appropriate deallocation behaviour. However, such non-reference counted pointers should be used *with extreme caution.* In particular, as the remoting infrastructure cannot know about the semantic relationship in use here, such back-pointers cannot be remote references.

The following rules call out common non-exceptions to Rule 1.

Rule 1a: *In-Out-parameters to functions.* The caller must AddRef the actual parameter, since it will be Released by the callee when the out-value is stored on top of it.

Rule 1b: *Fetching a global variable.* The local copy of the interface pointer fetched from an existing copy of the pointer in a global variable must be independently reference counted since called functions might destroy the copy in the global while the local copy is still alive.

Rule 1c: *New pointers synthesized out of "thin air."* A function which synthesizes an interface pointer using special internal knowledge rather than obtaining it from some other source must do an initial AddRef on the newly synthesized pointer. Important examples of such routines include instance creation routines, implementations of IUnknown::QueryInterface, etc.

Rule 1d: *Returning a copy of an internally stored pointer.* Once the pointer has been returned, the callee has no idea how its lifetime relates to that of the internally stored copy of the pointer. Thus, the callee must call AddRef on the pointer copy before returning it.

Rule 1e: *Call AddRef and Release on the same pointer.* Callee must make sure that the AddRef and the Release functions are called on the same interface pointer.

Finally, when implementing or using reference counted objects, a technique sometimes termed "artificial reference counts" sometimes proves useful. Suppose you're writing the code in method Foo in some interface IInterface. If in the implementation of Foo you invoke functions which have even the remotest chance of decrementing your

reference count, then such function may cause you to release before it returns to Foo. The subsequent code in Foo will crash.

A robust way to protect yourself from this is to insert an AddRef at the beginning of Foo which is paired with a Release just before Foo returns:

```
void IInterface::Foo(void)
{
	this->AddRef();

	// Body of Foo, as before, except short-circuit returnsneed to be changed.

	this->Release();
	return;
}
```

These "artificial" reference counts guarantee object stability while processing is done. Reference counting is done per interface basis to:

Simplify component debugging – had the reference counting is done at the component level only, determining alive and released interface had been complicated.
Facilitate better memory management – interface functions might keep holding the resources such as bitmaps images, audio files, buffers, etc. if the component lifetime is not maintained at the interface level.

Page intentionally left blank

3. 100% COM Program

Now let's look at our first COM program that conforms to the rules and guidelines of COM seen in chapter 1 and 2.

3.1 COM DLL Server Implementation

Program structure:

Server:

1. ClassFactoryDllServerWithRegFile.h – to declare public interface consumable by clients
2. ClassFactoryDllServerWithRegFile.cpp – to implement public and private operations
3. ClassFactoryDllServerWithRegFile.def (replacement to __declspec (dllexport)) - Module-definition (.def) files provide the linker with information about exports, attributes, and other information about the program to be linked. A .def file is most useful when building a DLL. DEF files allow exported symbols to be aliased to different imported symbols. Given enough time and information about each compiler's mangling scheme, the library vendor can produce a custom import library for each compiler. While tedious, this allows any compiler to gain link level compatibility with the DLL given a proper DEF file. Thus, link compatibility problems could be solved by providing module definition files for every possible compiler.

Client –

1. ClassFactoryDllServerWithRegFile.h – DLL server's header file to get public interfaces
2. ClientOfClassFactoryDllServerWithRegFile.cpp – client implementation
3. RegisterServer.reg - To create a scripting solution that changes a registry setting on a computer, you can use the registry editor to add or modify a registry setting and then export the setting to a .reg file. For us, .reg is used to register our COM-DLL program into the registry for COM engine to invoke at runtime.

COM server execution contexts -

CLSCTX_INPROC_SERVER - The code that creates and manages objects of this class runs in the same process as the caller of the function specifying the class context.
CLSCTX_INPROC_HANDLER - The code that manages objects of this class is an in-process handler. This is a dynamic-link library (DLL) that runs in the client process and implements client-side structures of this class when instances of the class are accessed remotely.

CLSCTX_LOCAL_SERVER - The EXE code that creates and manages objects of this class is loaded in a separate process space, which runs on same machine but in a different process.
CLSCTX_REMOTE_SERVER - A remote machine context. The LocalServer32 or LocalService code that creates and manages objects of this class is run on a different machine.

3.1.1 Server Header File

ClassFactoryDllServerWithRegFile.h

```
#pragma once

class ISum :public IUnknown
{
public:
   // ISum specific method declarations
   virtual HRESULT __stdcall SumOfTwoIntegers(int, int, int *) = 0;// pure virtual
};

class ISubtract :public IUnknown
{
public:
   // ISubtract specific method declarations
   virtual HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *) = 0;// pure virtual
};
```

```
// CLSID of SumSubtract Component {DFB1278D-20D3-4388-A83D-B2F2DE4EF59B}

const CLSID CLSID_SumSubtract =
{0xdfb1278d,0x20d3,0x4388,0xa8,0x3d,0xb2,0xf2,0xde,0x4e,0xf5,0x9b};

// IID of ISum Interface
const IID IID_ISum = {0x791876b8,0x4bd,0x4202,0x91,0x8d,0xc2,0x66,0x30,0x96,0xfe,0xbf};

// IID of ISubtract Interface
const IID IID_ISubtract = {0x9f2a8316,0x4eda,0x4113,0xac,0x4c,0x64,0x52,0x24,0x18,0x78,0x47};
```

3.1.2 Server Exported Functions Definition File

ClassFactoryDllServerWithRegFile.def

```
LIBRARY ClassFactoryDllServerWithRegFile
EXPORTS
      DllGetClassObject    @100 PRIVATE
      DllCanUnloadNow      @101 PRIVATE
```

/* Why PRIVATE? – Exported functions listed in the DEF file are added in the .LIB (exported method table) section of the DLL and thus can be invoked by the client statically or dynamically using GetDefProc(..) mechanism. To avoid such invocation - which produces unusual behaviours – and allow only COM engine to invoke these methods, make them PRIVATE. */

3.1.3 Server Source CPP File

ClassFactoryDllServerWithRegFile.cpp

```
#define UNICODE
#include<windows.h>
#include"ClassFactoryDllServerWithRegFile.h"

// class declarations
class CSumSubtract :public ISum, ISubtract
{
private:
   long m_cRef;
public:
   // constructor method
   declarations CSumSubtract(void);
   // destructor method declarations
   ~CSumSubtract(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // ISum specific method declarations (inherited)
   HRESULT __stdcall SumOfTwoIntegers(int, int, int *);

   // ISubtract specific method declarations (inherited) HRESULT
   __stdcall SubtractionOfTwoIntegers(int, int, int *);
};

class CSumSubtractClassFactory :public IClassFactory
{
private:
   long m_cRef;
public:
   // constructor method declarations
   CSumSubtractClassFactory(void);

   // destructor method declarations
   ~CSumSubtractClassFactory(void);
```

```
    // IUnknown specific method declarations (inherited)
    HRESULT __stdcall QueryInterface(REFIID, void **);
    ULONG __stdcall AddRef(void);
    ULONG __stdcall Release(void);

    // IClassFactory specific method declarations (inherited)
    HRESULT __stdcall CreateInstance(IUnknown *, REFIID, void
    **); HRESULT __stdcall LockServer(BOOL);
};

// global variable declarations
long glNumberOfActiveComponents = 0;    // number of active components
long glNumberOfServerLocks = 0;    // number of locks on this dll

// DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
    switch (dwReason)
    {
          case DLL_PROCESS_ATTACH:
             break;
          case DLL_PROCESS_DETACH:
             break;
    }

    return(TRUE);
}

// Implementation Of CSumSubtract's Constructor Method
CSumSubtract::CSumSubtract(void)
{
    // hardcoded initialization to anticipate possible failure of QueryInterface()
    m_cRef = 1;

    InterlockedIncrement(&glNumberOfActiveComponents);// increment global counter
}

// Implementation Of CSumSubtract's Destructor Method
CSumSubtract::~CSumSubtract(void)
{
    InterlockedDecrement(&glNumberOfActiveComponents);// decrement global counter
}

// Implementation Of CSumSubtract's IUnknown's Methods
HRESULT CSumSubtract::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
       *ppv = static_cast<ISum *>(this);
    else if (riid == IID_ISum)
       *ppv = static_cast<ISum *>(this);
    else if (riid == IID_ISubtract)
       *ppv = static_cast<ISubtract *>(this);
    else
    {
       *ppv = NULL;
       return(E_NOINTERFACE);
    }

    reinterpret_cast<IUnknown *>(*ppv)->AddRef();

    return(S_OK);
}

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

```
ULONG CSumSubtract::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of ISum's Methods
HRESULT CSumSubtract::SumOfTwoIntegers(int num1, int num2, int *pSum)
{
   *pSum = num1 + num2;

   return(S_OK);
}

// Implementation Of ISubtract's Methods
HRESULT CSumSubtract::SubtractionOfTwoIntegers(int num1, int num2, int *pSubtract)
{
   *pSubtract = num1 - num2;

   return(S_OK);
}

// Implementation Of CSumSubtractClassFactory's Constructor Method
CSumSubtractClassFactory::CSumSubtractClassFactory(void)
{
   m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()
}

// Implementation Of CSumSubtractClassFactory's Destructor Method
CSumSubtractClassFactory::~CSumSubtractClassFactory(void)
{
   // no code
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's IUnknown's Methods
HRESULT CSumSubtractClassFactory::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<IClassFactory *>(this);
   else if (riid == IID_IClassFactory)
      *ppv = static_cast<IClassFactory
   *>(this); else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

   reinterpret_cast<IUnknown *>(*ppv)->AddRef();

   return(S_OK);
}

ULONG CSumSubtractClassFactory::AddRef(void)
{
   InterlockedIncrement(&m_cRef);

   return(m_cRef);
}
```

```
ULONG CSumSubtractClassFactory::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's Methods
HRESULT CSumSubtractClassFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid, void **ppv)
{
   // variable declarations
   CSumSubtract *pCSumSubtract =
   NULL; HRESULT hr;

   // code
   if (pUnkOuter != NULL)
      return(CLASS_E_NOAGGREGATION);

   // create the instance of component i.e. of CSumSubtract
   class pCSumSubtract = new CSumSubtract;

   if (pCSumSubtract == NULL)
      return(E_OUTOFMEMORY);

   // get the requested interface
   hr = pCSumSubtract->QueryInterface(riid, ppv);

   pCSumSubtract->Release();// anticipate possible failure of QueryInterface()
   return(hr);
}

HRESULT CSumSubtractClassFactory::LockServer(BOOL fLock)
{
   if (fLock)
      InterlockedIncrement(&glNumberOfServerLocks);
   else
      InterlockedDecrement(&glNumberOfServerLocks);

   return(S_OK);
}

// Implementation Of Exported Functions From This Dll
HRESULT __stdcall DllGetClassObject(REFCLSID rclsid, REFIID riid, void **ppv)
{
   // variable declaraions
   CSumSubtractClassFactory *pCSumSubtractClassFactory = NULL;
   HRESULT hr;

   // code
   if (rclsid != CLSID_SumSubtract)
      return(CLASS_E_CLASSNOTAVAILABLE);

   // create class factory
   pCSumSubtractClassFactory = new CSumSubtractClassFactory; if
   (pCSumSubtractClassFactory == NULL)
      return(E_OUTOFMEMORY);

   hr = pCSumSubtractClassFactory->QueryInterface(riid, ppv); pCSumSubtractClassFactory-
   >Release();// anticipate possible failure of QueryInterface()

   return(hr);
}
```

```
HRESULT __stdcall DllCanUnloadNow(void)
{
   if ((glNumberOfActiveComponents == 0) && (glNumberOfServerLocks == 0))
      return(S_OK);
   else
      return(S_FALSE);
}
```

3.2 Client Implementation

3.2.1 Client Header File

As discussed earlier, client obtains the header file from the server developer to get CoClass and its interfaces related information. Hence, in our case the header file is same as given in the section 3.1.1 i.e. **ClassFactoryDllServerWithRegFile.h**.

3.2.2 Server Registration File

ClassFactoryDllServerWithRegFile.reg

// If DLL server is 32-bit on 64-bit OS, then register the DLL under virtual registry node WOW6432Node

```
REGEDIT4
[HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{DFB1278D-20D3-4388-A83D-B2F2DE4EF59B}]
@="MyComDll"
[HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{DFB1278D-20D3-4388-A83D-B2F2DE4EF59B}\InprocServer32]
@="e:\\x86\\ClassFactoryDllServerWithRegFile.dll"
```

// If the DLL server is 32-bit on 32-bit OS or the server is 64-bit then register the DLL under native registry node

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{DFB1278D-20D3-4388-A83D-B2F2DE4EF59B}]
@="MyComDll"
[HKEY_CLASSES_ROOT\CLSID\{DFB1278D-20D3-4388-A83D-B2F2DE4EF59B}\InprocServer32]
@="e:\\x64\\ClassFactoryDllServerWithRegFile.dll"
```

3.2.3 Self-Registering DLL's

In-process COM servers support self-registration through several DLL entry points with well-known names. The DLL entry points for registering and unregistering a server are defined as follows:

`HRESULT DllRegisterServer(void);` - Instructs an in-process server to create its registry entries for all classes supported in this server module.
`HRESULT DllUnregisterServer(void);` - Instructs an in-process server to remove only those entries created through DllRegisterServer.

Both of these entry points are required for a DLL to be self-registering. The implementation of the DllRegisterServer entry point adds or updates registry information for all the classes implemented by the DLL. The DllUnregisterServer entry point removes its information from the registry.

To register a module, refer to following instructions:

1. Export DllRegisterServer and DllUnregisterServer methods. Your final def file will look like this

```
LIBRARY ClassFactoryDllServerWithRegFile
EXPORTS
    DllRegisterServer   @100 PRIVATE
    DllUnregisterServer @101 PRIVATE
    DllGetClassObject   @102 PRIVATE
    DllCanUnloadNow     @103 PRIVATE
```

2. Implement DllRegisterServer method, sample implementation follows

```
STDAPI DllRegisterServer()
{
	HKEY  hCLSIDKey = NULL, hInProcSvrKey = NULL;
	LONG  lRet;
	TCHAR szModulePath[MAX_PATH];
	TCHAR szClassDescription[] = TEXT("Simple COM class");
	TCHAR szThreadingModel[] = TEXT("Apartment");

	__try
	{
	 // Create a key under CLSID for our COM server.

	 // 32bit server on 32bit OS -> HKEY_CLASSES_ROOT\CLSID\{DFB1278D-20D3-4388-
	 // A83D-B2F2DE4EF59B}
	 // 32bit server on 64bit OS -> HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{DFB1278D-
	 // 20D3-4388-A83D-B2F2DE4EF59B} 64bit server on 32bit OS -> N/A
	 // 64bit server on 64bit OS -> HKEY_CLASSES_ROOT\CLSID\{DFB1278D-20D3-4388-
	 // A83D-B2F2DE4EF59B}

	 lRet = RegCreateKeyEx(HKEY_CLASSES_ROOT, TEXT("CLSID\\{DFB1278D-20D3-4388-A83D-
	 B2F2DE4EF59B}"), 0, NULL, REG_OPTION_NON_VOLATILE, KEY_SET_VALUE |
	 KEY_CREATE_SUB_KEY, NULL, &hCLSIDKey, NULL);

	 if (ERROR_SUCCESS != lRet)
	    return HRESULT_FROM_WIN32(lRet);

	 // The default value of the key is a human-readable description of the coclass.

	 lRet = RegSetValueEx(hCLSIDKey, NULL, 0, REG_SZ, (const
	 BYTE*)szClassDescription, sizeof(szClassDescription));

	 if (ERROR_SUCCESS != lRet)
	    return HRESULT_FROM_WIN32(lRet);

	 // Create the InProcServer32 key, which holds info about our coclass.

	 lRet = RegCreateKeyEx(hCLSIDKey, TEXT("InProcServer32"), 0, NULL,
	 REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hInProcSvrKey, NULL);

	 if (ERROR_SUCCESS != lRet)
	    return HRESULT_FROM_WIN32(lRet);

	 // The default value of the InProcServer32 key holds the full path to our DLL.
	 // Global ghModule (HMODULE) variable is assigned to current DLL handle in
	 // DllMain->DLL_PROCESS_ATTACH case

	 GetModuleFileName(ghModule, szModulePath, MAX_PATH);

	 lRet = RegSetValueEx(hInProcSvrKey, NULL, 0, REG_SZ, (const
	    BYTE*)szModulePath, sizeof(TCHAR) * (lstrlen(szModulePath) + 1));

	 if (ERROR_SUCCESS != lRet)
	    return HRESULT_FROM_WIN32(lRet);

	 // The ThreadingModel value tells COM how it should handle threads in our DLL.

	 lRet = RegSetValueEx(hInProcSvrKey, TEXT("ThreadingModel"), 0, REG_SZ,
	    (const BYTE*)szThreadingModel, sizeof(szThreadingModel));

	 if (ERROR_SUCCESS != lRet)
	    return HRESULT_FROM_WIN32(lRet);
	}
	__finally
```

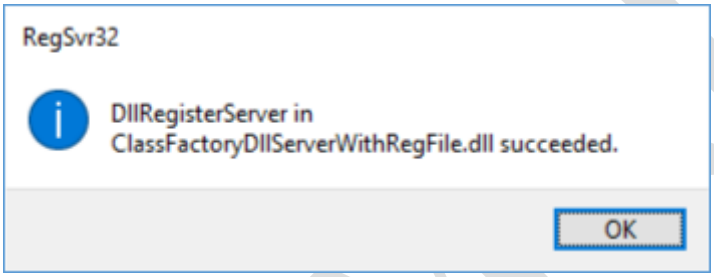
```
        {
          if (NULL != hCLSIDKey)
             RegCloseKey(hCLSIDKey);

          if (NULL != hInProcSvrKey)
             RegCloseKey(hInProcSvrKey);
        }

        return S_OK;
}
```

3. Implement DllUnregisterServer method, sample implementation follows

```
STDAPI DllUnregisterServer()
{
      RegDeleteKey(HKEY_CLASSES_ROOT, TEXT("CLSID\\{DFB1278D-20D3-4388-A83D-
      B2F2DE4EF59B}\\InProcServer32"));
      RegDeleteKey(HKEY_CLASSES_ROOT, TEXT("CLSID\\{DFB1278D-20D3-4388-A83D-
      B2F2DE4EF59B}"));

      return S_OK;
}
```

4. Register the module from command (elevated admin mode) line using regsvr32 tool:

```
regsvr32 <binary name> i.e. regsvr32 ClassFactoryDllServerWithRegFile.dll
```



3.2.4 Client Source CPP File

ClientOfClassFactoryDllServerWithRegFile.cpp

```
#define UNICODE
#include<windows.h>
#include"ClassFactoryDllServerWithRegFile.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// global variable declarations
ISum *pISum = NULL; ISubtract
*pISubtract = NULL;

// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
   // variable
   declarations WNDCLASSEX
   wndclass; HWND hwnd;
   MSG msg;
   TCHAR AppName[] =
   TEXT("ComClient"); HRESULT hr;

   // COM Initialization
   hr = CoInitialize(NULL);

   if (FAILED(hr))
```

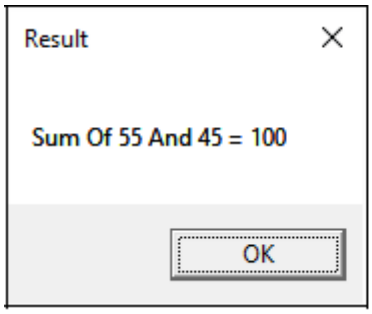
```
    {
        MessageBox(NULL, TEXT("COM Library Can Not Be Initialized.\nProgram Will Now Exit."),
        TEXT("Program Error"), MB_OK);
        exit(0);
    }

    // WNDCLASSEX initialization wndclass.cbSize =
    sizeof(wndclass); wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.cbClsExtra = 0; wndclass.cbWndExtra = 0;
    wndclass.lpfnWndProc = WndProc; wndclass.hIcon =
    LoadIcon(NULL, IDI_APPLICATION); wndclass.hCursor =
    LoadCursor(NULL, IDC_ARROW); wndclass.hbrBackground =
    (HBRUSH)GetStockObject(WHITE_BRUSH); wndclass.hInstance =
    hInstance;


    wndclass.lpszClassName = AppName;
    wndclass.lpszMenuName = NULL;
    wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // register window class
    RegisterClassEx(&wndclass);

    // create window
    hwnd = CreateWindow(AppName, TEXT("Client Of COM Dll Server"),
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL);

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // COM Un-initialization
    CoUninitialize();
    return((int)msg.wParam);
}

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    // function declarations
    void SafeInterfaceRelease(void);

    // variable declarations
    HRESULT hr;
    int iNum1, iNum2,
    iSum; TCHAR str[255];

    // code
    switch (iMsg)
    {
    case WM_CREATE:
        // CoCreateInstance works with only one interface at a time. To avoid network
        // traffic, use CoCreateInstanceEx API to request for multiple interface pointers
        // in one network round-trip. Each interface request can be specified using
```

```
        // a MULTI_QI structure. CoCreateInstanceEx API is also used to get remote COM
        // server interface.

        hr = CoCreateInstance(CLSID_SumSubtract, NULL, CLSCTX_INPROC_SERVER, IID_ISum,
        (void **)&pISum);

        if (FAILED(hr))
        {
           MessageBox(hwnd, TEXT("ISum Interface Can Not Be Obtained"), TEXT("Error"), MB_OK);
           DestroyWindow(hwnd);
        }

        // initialize arguments
        hardcoded iNum1 = 55;
        iNum2 = 45;

        // call SumOfTwoIntegers() of ISum to get the sum
        pISum->SumOfTwoIntegers(iNum1, iNum2, &iSum);

        // display the result
        wsprintf(str, TEXT("Sum Of %d And %d = %d"), iNum1, iNum2, iSum);
        MessageBox(hwnd, str, TEXT("Result"), MB_OK);

        // call QueryInterface() on ISum,to get ISubtract's pointer
        hr = pISum->QueryInterface(IID_ISubtract, (void **)&pISubtract);

        if (FAILED(hr))
        {
           MessageBox(hwnd, TEXT("ISubtract Interface Can Not Be Obtained"), TEXT("Error"),
           MB_OK);
           DestroyWindow(hwnd);
        }

        // as ISum is now not needed onwords, release
        it pISum->Release();
        pISum = NULL;// make relesed interface NULL
                // again initialize arguments hardcoded
        iNum1 = 155;
        iNum2 = 145;

        // again call SumOfTwoIntegers() of ISum to get the new sum
        pISubtract->SubtractionOfTwoIntegers(iNum1, iNum2, &iSum);

        // as ISum is now not needed onwords, release it
        pISubtract->Release();
        pISubtract = NULL;// make relesed interface NULL
                    // display the result
        wsprintf(str, TEXT("Subtraction Of %d And %d = %d"), iNum1, iNum2,
        iSum); MessageBox(hwnd, str, TEXT("Result"), MB_OK);

        // exit the application
        DestroyWindow(hwnd);
        break;
    case WM_DESTROY:
        // relinquish the interfaces
        SafeInterfaceRelease();

        PostQuitMessage(0);
        break;
    }
    return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}

// Method is a safety measure to release interfaces if not previously
void SafeInterfaceRelease(void)
{
    // code
```

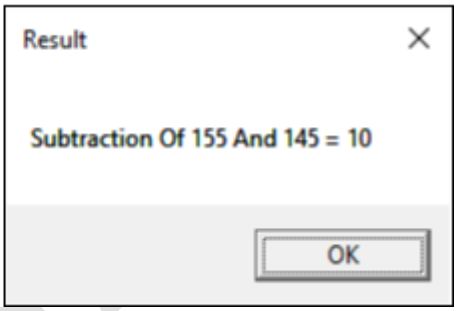
```
    if (pISum)
    {
        pISum->Release();
        pISum = NULL;
    }

    if (pISubtract)
    {
        pISubtract->Release();
        pISubtract = NULL;
    }
}
```

After running the client you will get output as below:

Output of SumOfTwoIntegers call:



Output of SubtractionOfTwoIntegers call:



3.3 How client gets interface pointer?

1

Client calls CoCreateInstance (CLSID_SumSubtract, NULL, CLSCTX_INPROC_SERVER, IID_ISum, (void **)&pISum);

- CLSID_SumSubtract: CLSID i.e. identifier of the CoClass
- NULL: Not an aggregated component
- CLSCTX_INPROC_SERVER: In process server
- IID_ISum: ISum IID i.e. identifier of interface required
- (void **)&pISum: pointer to interface ISum

2

The COM engine internally calls CoGetClassObject API with following parameters:

- CLSID of component: i.e. 1st parameter of CoCreateInstance.
- Context: i.e. 3rd parameter of CoCreateInstance.
- NULL (as parameter CoServerInfo is valid for DCOM only)
- IID_IClassFactory
- NULL: The address of pointer variable that receives the interface pointer requested in *riid*. Upon successful return, **ppv* contains the requested interface pointer

3

CoGetClassObject refers to its 2nd parameter to find out the execution context i.e. inproc, outproc, remote. For inproc COM server, it looks for the CLSID in the system registry. If found then it reads the corresponding value of the inproc subkey to get the absolute server path. Having the server name, next CoGetClassObject calls CoLoadLibrary with the server path and flag (true) to load the server using LoadLibrary call. CoFreeUnusedLibrary also called periodically to free the library server, if not in use.

4

CoGetClassObject now loads server's DllGetClassObject exported method using GetProcAddress WIN32 API. If loaded successfully then DllGetClassObject function is called with following 3 arguments:

1. server's CLSID
2. IID_IClassFactory
3. NULL

5

DllGetClassObject verifies that the CLSID is correct. Once verified, an instance of the ClassFactory class is created followed by a call to the QueryInterface method of the ClassFactory object to get IClassFactory Interface pointer. CoGetClassObject then passes this interface pointer back to CoCreateInstance.

6

CoCreateInstance now calls CreateInstance method of the IClassFactory interface with following arguments:

1. PUnkOuter
2. IID of the required interface
3. interface pointer (NULL initialized)

CreateInstance creates an instance of the server component followed by a call to the QueryInteface on that component to get the desired interface pointer. If call succeeds, a valid interface pointer is returned using the interface-pointer argument passed in to the CoCreateInstance method. Finally, CoCreateInstance releases the ClassFactory and returns interface pointer to the client.

Page intentionally left blank

Page intentionally left blank

4. Object Reusability

An important goal of any object model is that component objects can be reused and extended at binary level. Implementation inheritance is one way to achieve this. Code can be reused to build a new object and you need to inherit implementation from it and override methods in the traditional C++ way. However, this way of object reuse is simply not robust enough for large, evolving systems composed of software components. For this reason, COM introduces other reusability mechanisms:

The key point to building reusable components is black-box reuse which means the piece of code attempting to reuse another component knows nothing, and does not need to know anything, about the internal structure or implementation of the component being used. In other words, the code attempting to reuse a component depends upon the behaviour of the component and not the exact implementation.

To achieve black-box reusability, COM supports two mechanisms through which one object may reuse another. For convenience, the object being reused is called the "inner object" and the object making use of that inner object is the "outer object."

4.1 Containment/Delegation

The outer object behaves like an object client to the inner object. The outer object "contains" the inner object and when the outer object wishes to use the services of the inner object the outer object simply delegates implementation to the inner object's interfaces. In other words, the outer object uses the inner's services to implement itself. It is not necessary that the outer and inner objects support the same interfaces; in fact, the outer object may use an inner object's interface to help implement parts of a different interface on the outer object especially when the complexity of the interfaces differs greatly.

As far as the clients are concerned, both objects implement interfaces A, B, and C. Furthermore, the client treats the outer object as a black box, and thus does not care, nor does it need to care, about the internal structure of the outer object. The client only cares about behaviour.

Containment is simple to implement for an outer object: during its creation, the outer object creates whatever inner objects it needs to use as any other client would.

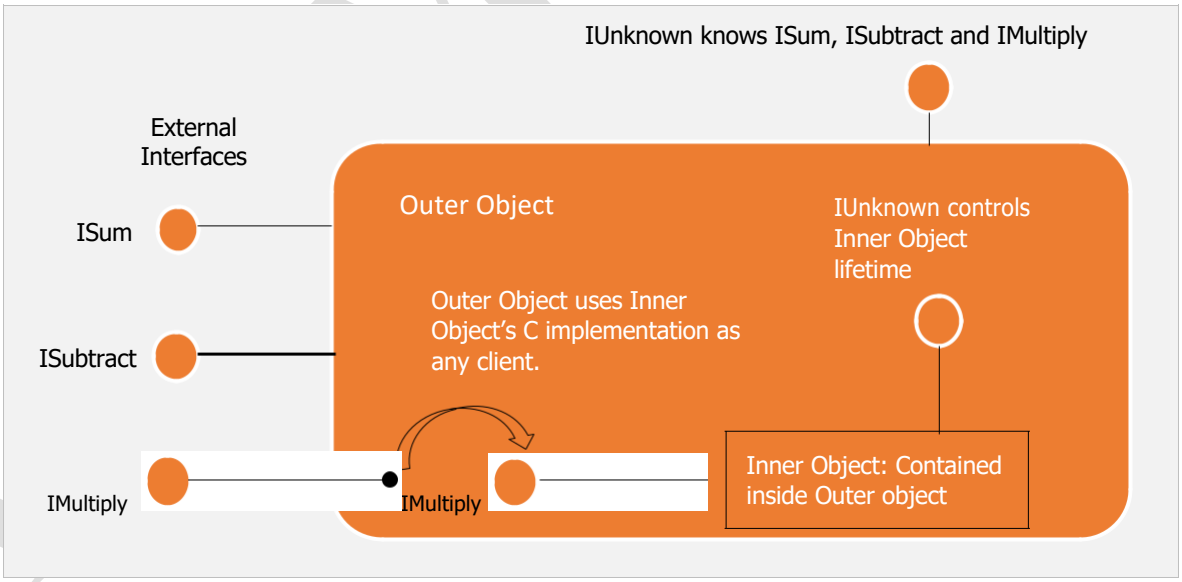


Figure 4.1: Containment of an inner object and delegation to its interfaces

4.1.1 Server/Client Implementation

Program structure:

Containment Server:

Inner component:

1. ContainmentInnerComponentWithRegFile.h – to declare public interface consumable by clients (inner component exposed interfaces)
2. ContainmentInnerComponentWithRegFile.cpp – to implement/define public and private operations for inner component
3. ContainmentInnerComponentWithRegFile.def – inner component's exported methods

Outer component:

1. ContainmentOuterComponentWithregFile.h – to declare public interface consumable by clients (outer component exposed interfaces)
2. ContainmentInnerComponentWithRegFile.h – to declare public interface consumable by clients (inner component exposed interfaces)
3. ContainmentOuterComponentWithRegFile.cpp – to implement public and private operations for outer component
4. ContainmentOuterComponentWithRegFile.def – outer component's exported methods

Containment Client –

1. HeaderForClientOfComponentWithRegFile.h – DLL server's header file to get public interfaces (inner and outer components' – merged together)
2. ClientOfContainmentWithRegFile.cpp – client implementation to use outer component
3. RegisterServer.reg – registry file to register inner and outer components into the system registry

4.1.1.1 Containment Inner Component

ContainmentInnerComponentWithRegFile.h

```
#pragma once

class IMultiplication :public IUnknown
{
public:
   // IMultiplication specific method declarations pure virtual
   virtual HRESULT __stdcall MultiplicationOfTwoIntegers(int, int, int *) = 0};

class IDivision :public IUnknown
{
public:
   // IDivision specific method declarations
   virtual HRESULT __stdcall DivisionOfTwoIntegers(int, int, int *) = 0; // pure virtual
};

// CLSID of MultiplicationDivision Component {764E7C57-1737-4f19-927A-6081C39EF514}
const CLSID CLSID_MultiplicationDivision = {
0x764e7c57,0x1737,0x4f19,0x92,0x7a,0x60,0x81,0xc3,0x9e,0xf5,0x14 };

// IID of IMultiplication Interface
const IID IID_IMultiplication = {
0xa39f8306,0x7a0c,0x4e47,0xb3,0x8a,0xfc,0x8d,0x68,0x5d,0xca,0x90 };

// IID of IDivision Interface
const IID IID_IDivision = { 0x9f7d9e5a,0xab6,0x4a59,0xad,0x22,0xa,0x89,0x88,0x2e,0x46,0x28 };
```

ContainmentInnerComponentWithRegFile.def

```
LIBRARY ContainmentInnerComponentWithRegFile
EXPORTS
      DllGetClassObject   PRIVATE
      DllCanUnloadNow     PRIVATE
```

ContainmentInnerComponentWithRegFile.cpp

```
#include <windows.h>
#include"ContainmentInnerComponentWithRegFile.h"

// class declarations
class CMultiplicationDivision :public IMultiplication, IDivision
{
private:
   long m_cRef;
public:
   // constructor method declarations
   CMultiplicationDivision(void);
   // destructor method declarations
   ~CMultiplicationDivision(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // IMultiplication specific method declarations (inherited)
   HRESULT __stdcall MultiplicationOfTwoIntegers(int, int, int *);

   // IDivision specific method declarations (inherited)
   HRESULT __stdcall DivisionOfTwoIntegers(int, int, int *);
};

class CMultiplicationDivisionClassFactory :public IClassFactory
{
private:
   long m_cRef;
public:
   // constructor method declarations
   CMultiplicationDivisionClassFactory(void);
   // destructor method declarations
   ~CMultiplicationDivisionClassFactory(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // IClassFactory specific method declarations (inherited) HRESULT
   __stdcall CreateInstance(IUnknown *, REFIID, void **);

   HRESULT __stdcall LockServer(BOOL);
};

// global variable declarations
long glNumberOfActiveComponents = 0;// number of active components
long glNumberOfServerLocks = 0;// number of locks on this dll

// DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
   switch (dwReason)
   {
   case DLL_PROCESS_ATTACH:
```

```
      break;
   case DLL_PROCESS_DETACH:
      break;
   }

   return(TRUE);
}

// Implementation Of CMultiplicationDivision's Constructor Method
CMultiplicationDivision::CMultiplicationDivision(void)
{
   m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()
   InterlockedIncrement(&glNumberOfActiveComponents);// increment global counter
}

// Implementation Of CSumSubtract's Destructor Method
CMultiplicationDivision::~CMultiplicationDivision(void)
{
   InterlockedDecrement(&glNumberOfActiveComponents);// decrement global counter
}

// Implementation Of CMultiplicationDivision's IUnknown's Methods
HRESULT CMultiplicationDivision::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<IMultiplication *>(this);
   else if (riid == IID_IMultiplication)
      *ppv = static_cast<IMultiplication
   *>(this); else if (riid == IID_IDivision)
      *ppv = static_cast<IDivision
   *>(this); else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

   reinterpret_cast<IUnknown *>(*ppv)->AddRef();

   return(S_OK);
}

ULONG CMultiplicationDivision::AddRef(void)
{
   InterlockedIncrement(&m_cRef);
   return(m_cRef);
}

ULONG CMultiplicationDivision::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of IMultiplication's Methods
HRESULT CMultiplicationDivision::MultiplicationOfTwoIntegers(int num1, int num2, int
*pMultiplication)
{
   *pMultiplication = num1*num2;
   return(S_OK);
}
```

```
// Implementation Of IDivision's Methods
HRESULT CMultiplicationDivision::DivisionOfTwoIntegers(int num1, int num2, int *pDivision)
{
    *pDivision = num1 / num2;
    return(S_OK);
}

// Implementation Of CMultiplicationDivisionClassFactory's Constructor Method
CMultiplicationDivisionClassFactory::CMultiplicationDivisionClassFactory(void)
{
    m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()
}

// Implementation Of CMultiplicationDivisionClassFactory's Destructor Method
CMultiplicationDivisionClassFactory::~CMultiplicationDivisionClassFactory(void)
{
    // no code
}

// Implementation Of CMultiplicationDivisionClassFactory's IClassFactory's IUnknown's Methods
HRESULT CMultiplicationDivisionClassFactory::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IClassFactory *>(this);
    else if (riid == IID_IClassFactory)
        *ppv = static_cast<IClassFactory
    *>(this); else
    {
        *ppv = NULL;
        return(E_NOINTERFACE);
    }
    reinterpret_cast<IUnknown *>(*ppv)->AddRef();

    return(S_OK);
}

ULONG CMultiplicationDivisionClassFactory::AddRef(void)
{
    InterlockedIncrement(&m_cRef);
    return(m_cRef);
}

ULONG CMultiplicationDivisionClassFactory::Release(void)
{
    InterlockedDecrement(&m_cRef);

    if (m_cRef == 0)
    {
        delete(this);
        return(0);
    }

    return(m_cRef);
}

// Implementation Of CMultiplicationDivisionClassFactory's IClassFactory's Methods
HRESULT CMultiplicationDivisionClassFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid,
void **ppv)
{
    // variable declarations
    CMultiplicationDivision *pCMultiplicationDivision =
    NULL; HRESULT hr;

    if (pUnkOuter != NULL)
        return(CLASS_E_NOAGGREGATION);

    // create the instance of component i.e. of CMultiplicationDivision
    class pCMultiplicationDivision = new CMultiplicationDivision;
```

```
    if (pCMultiplicationDivision == NULL)
        return(E_OUTOFMEMORY);

    // get the requested interface
    hr = pCMultiplicationDivision->QueryInterface(riid, ppv);

    pCMultiplicationDivision->Release();// anticipate possible failure of
    QueryInterface() return(hr);
}

HRESULT CMultiplicationDivisionClassFactory::LockServer(BOOL fLock)
{
    if (fLock)
        InterlockedIncrement(&glNumberOfServerLocks);
    else
        InterlockedDecrement(&glNumberOfServerLocks);

    return(S_OK);
}

// Implementation Of Exported Functions From This Dll
HRESULT __stdcall DllGetClassObject(REFCLSID rclsid, REFIID riid, void **ppv)
{
    // variable declaraions
    CMultiplicationDivisionClassFactory *pCMultiplicationDivisionClassFactory =
    NULL; HRESULT hr;

    if (rclsid != CLSID_MultiplicationDivision)
        return(CLASS_E_CLASSNOTAVAILABLE);

    // create class factory
    pCMultiplicationDivisionClassFactory = new CMultiplicationDivisionClassFactory;

    if (pCMultiplicationDivisionClassFactory == NULL)
        return(E_OUTOFMEMORY);

    hr = pCMultiplicationDivisionClassFactory->QueryInterface(riid, ppv);

        // anticipate possible failure of QueryInterface()
    pCMultiplicationDivisionClassFactory->Release();
        return(hr);
}

HRESULT __stdcall DllCanUnloadNow(void)
{
    if ((glNumberOfActiveComponents == 0) && (glNumberOfServerLocks == 0))
        return(S_OK);
    else
        return(S_FALSE);
}
```

4.1.1.2 Containment Outer Component

ContainmentInnerComponentWithRegFile.h – refer to containment inner component's header file

ContainmentOuterComponentWithregFile.h

```
#pragma once

class ISum :public IUnknown
{
public:
    // ISum specific method declarations
    virtual HRESULT __stdcall SumOfTwoIntegers(int, int, int *) = 0;// pure virtual
};
```

```
class ISubtract :public IUnknown
{
public:
   // ISubtract specific method declarations
   virtual HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *) = 0;// pure virtual
};

// CLSID of SumSubtract Component {6D9B9F18-1DD6-4377-946A-5ED883B78031}
const CLSID CLSID_SumSubtract = {
0x6d9b9f18,0x1dd6,0x4377,0x94,0x6a,0x5e,0xd8,0x83,0xb7,0x80,0x31 };

// IID of ISum Interface
const IID IID_ISum = { 0x791876b8,0x4bd,0x4202,0x91,0x8d,0xc2,0x66,0x30,0x96,0xfe,0xbf };

// IID of ISubtract Interface
const IID IID_ISubtract = { 0x9f2a8316,0x4eda,0x4113,0xac,0x4c,0x64,0x52,0x24,0x18,0x78,0x47 };
```

ContainmentOuterComponentWithRegFile.def

```
LIBRARY ContainmentOuterComponentWithRegFile
EXPORTS
      DllGetClassObject    PRIVATE
      DllCanUnloadNow      PRIVATE
```

ContainmentOuterComponentWithRegFile.cpp

```
#include<windows.h>
#include"ContainmentInnerComponentWithRegFile.h"
#include"ContainmentOuterComponentWithRegFile.h"

// class declarations
class CSumSubtract :public ISum, ISubtract, IMultiplication, IDivision
{
private:
   long m_cRef;
   IMultiplication *m_pIMultiplication;
   IDivision *m_pIDivision;
public:
   // constructor method
   declarations CSumSubtract(void);
   // destructor method declarations
   ~CSumSubtract(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // ISum specific method declarations (inherited)
   HRESULT __stdcall SumOfTwoIntegers(int, int, int *);

   // ISubtract specific method declarations (inherited) HRESULT
   __stdcall SubtractionOfTwoIntegers(int, int, int *);

   // IMultiplication specific method declarations (inherited)
   HRESULT __stdcall MultiplicationOfTwoIntegers(int, int, int *);

   // IDivision specific method declarations (inherited)
   HRESULT __stdcall DivisionOfTwoIntegers(int, int, int *);

   // custom method for inner component creation
   HRESULT __stdcall InitializeInnerComponent(void);
};
```

```
class CSumSubtractClassFactory :public IClassFactory
{
private:
   long m_cRef;
public:
   // constructor method declarations
   CSumSubtractClassFactory(void);
   // destructor method declarations
   ~CSumSubtractClassFactory(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // IClassFactory specific method declarations (inherited) HRESULT
   __stdcall CreateInstance(IUnknown *, REFIID, void **);

   HRESULT __stdcall LockServer(BOOL);
};

// global variable declarations
long glNumberOfActiveComponents = 0;// number of active components
long glNumberOfServerLocks = 0;// number of locks on this dll
                        // DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
   // code
   switch (dwReason)
   {
         case DLL_PROCESS_ATTACH:
            break;
         case DLL_PROCESS_DETACH:
            break;
   }

   return(TRUE);
}

// Implementation Of CSumSubtract's Constructor Method
CSumSubtract::CSumSubtract(void)
{
   // initialization of private data members
   m_pIMultiplication = NULL;
   m_pIDivision = NULL;
   m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()

   InterlockedIncrement(&glNumberOfActiveComponents);// increment global counter
}

// Implementation Of CSumSubtract's Destructor Method
CSumSubtract::~CSumSubtract(void)
{
   InterlockedDecrement(&glNumberOfActiveComponents);// decrement global counter

   if (m_pIMultiplication)
   {
      m_pIMultiplication->Release();
      m_pIMultiplication = NULL;
   }

   if (m_pIDivision)
   {
      m_pIDivision->Release();
      m_pIDivision = NULL;
   }
}
```

```
// Implementation Of CSumSubtract's IUnknown's Methods
HRESULT CSumSubtract::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<ISum *>(this);
    else if (riid == IID_ISum)
        *ppv = static_cast<ISum *>(this);
    else if (riid == IID_ISubtract)
        *ppv = static_cast<ISubtract *>(this);
    else if (riid == IID_IMultiplication)
        *ppv = static_cast<IMultiplication
    *>(this); else if (riid == IID_IDivision)
        *ppv = static_cast<IDivision
    *>(this); else
    {
        *ppv = NULL;
        return(E_NOINTERFACE);
    }

    reinterpret_cast<IUnknown *>(*ppv)->AddRef();

    return(S_OK);
}

ULONG CSumSubtract::AddRef(void)
{
    InterlockedIncrement(&m_cRef);

    return(m_cRef);
}

ULONG CSumSubtract::Release(void)
{
    InterlockedDecrement(&m_cRef);

    if (m_cRef == 0)
    {
        delete(this);
        return(0);
    }

    return(m_cRef);
}

// Implementation Of ISum's Methods
HRESULT CSumSubtract::SumOfTwoIntegers(int num1, int num2, int *pSum)
{
    *pSum = num1 + num2;

    return(S_OK);
}

// Implementation Of ISubtract's Methods
HRESULT CSumSubtract::SubtractionOfTwoIntegers(int num1, int num2, int *pSubtract)
{
    *pSubtract = num1 - num2;
    return(S_OK);
}

// Implementation Of IMultiplication's Methods
HRESULT CSumSubtract::MultiplicationOfTwoIntegers(int num1, int num2, int *pMultiplication)
{
    // delegate to inner component
    m_pIMultiplication->MultiplicationOfTwoIntegers(num1, num2, pMultiplication);

    return(S_OK);
}
```

```
// Implementation Of IDivision's Methods
HRESULT CSumSubtract::DivisionOfTwoIntegers(int num1, int num2, int *pDivision)
{
   // delegate to inner component
   m_pIDivision->DivisionOfTwoIntegers(num1, num2, pDivision);
   return(S_OK);
}

HRESULT CSumSubtract::InitializeInnerComponent(void)
{
   // variable declarations
   HRESULT hr;

   hr = CoCreateInstance(CLSID_MultiplicationDivision, NULL, CLSCTX_INPROC_SERVER,
      IID_IMultiplication, (void **)&m_pIMultiplication);

   if (FAILED(hr))
   {
      MessageBox(NULL, TEXT("IMultiplication Interface Can Not Be Obtained From Inner
      Component."), TEXT("Error"), MB_OK);
      return(E_FAIL);
   }

   hr = m_pIMultiplication->QueryInterface(IID_IDivision, (void **)&m_pIDivision);

   if (FAILED(hr))
   {
      MessageBox(NULL, TEXT("IDivision Interface Can Not Be Obtained From Inner Component."),
      TEXT("Error"), MB_OK);
      return(E_FAIL);
   }

   return(S_OK);
}

// Implementation Of CSumSubtractClassFactory's Constructor Method
CSumSubtractClassFactory::CSumSubtractClassFactory(void)
{
   // hardcoded initialization to anticipate possible failure of
   QueryInterface() m_cRef = 1;
}

// Implementation Of CSumSubtractClassFactory's Destructor Method
CSumSubtractClassFactory::~CSumSubtractClassFactory(void)
{
   // no code
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's IUnknown's Methods
HRESULT CSumSubtractClassFactory::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<IClassFactory *>(this);
   else if (riid == IID_IClassFactory)
      *ppv = static_cast<IClassFactory
   *>(this); else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

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

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

ULONG CSumSubtractClassFactory::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's Methods
HRESULT CSumSubtractClassFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid, void **ppv)
{
   // variable declarations
   CSumSubtract *pCSumSubtract =
   NULL; HRESULT hr;

   if (pUnkOuter != NULL)
      return(CLASS_E_NOAGGREGATION);

   // create the instance of component i.e. of CSumSubtract
   class pCSumSubtract = new CSumSubtract;

   if (pCSumSubtract == NULL)
      return(E_OUTOFMEMORY);

   // initialize the inner component
   hr = pCSumSubtract->InitializeInnerComponent();

   if (FAILED(hr))
   {
      MessageBox(NULL, TEXT("Failed To Initialize Inner Component"), TEXT("Error"), MB_OK);
      pCSumSubtract->Release();
      return(hr);
   }

   // get the requested interface
   hr = pCSumSubtract->QueryInterface(riid, ppv);

   pCSumSubtract->Release();// anticipate possible failure of QueryInterface()
   return(hr);
}

HRESULT CSumSubtractClassFactory::LockServer(BOOL fLock)
{
   if (fLock)
      InterlockedIncrement(&glNumberOfServerLocks);
   else
      InterlockedDecrement(&glNumberOfServerLocks);

   return(S_OK);
}

// Implementation Of Exported Functions From This Dll
HRESULT __stdcall DllGetClassObject(REFCLSID rclsid, REFIID riid, void **ppv)
{
   // variable declaraions
   CSumSubtractClassFactory *pCSumSubtractClassFactory = NULL;
   HRESULT hr;
```

```
    if (rclsid != CLSID_SumSubtract)
        return(CLASS_E_CLASSNOTAVAILABLE);

    // create class factory
    pCSumSubtractClassFactory = new CSumSubtractClassFactory;

    if (pCSumSubtractClassFactory == NULL)
        return(E_OUTOFMEMORY);

    hr = pCSumSubtractClassFactory->QueryInterface(riid, ppv);

    pCSumSubtractClassFactory->Release();// anticipate possible failure of
    QueryInterface() return(hr);
}

HRESULT __stdcall DllCanUnloadNow(void)
{
    if ((glNumberOfActiveComponents == 0) && (glNumberOfServerLocks == 0))
        return(S_OK);
    else
        return(S_FALSE);
}
```

4.1.1.3 Containment Client

HeaderForClientOfComponentWithRegFile.h – Merge inner and outer components' header files together

RegisterServer.reg

```
[HKEY_CLASSES_ROOT\CLSID\{6D9B9F18-1DD6-4377-946A-5ED883B78031}]
@="OuterComDll_WithRegFile"
[HKEY_CLASSES_ROOT\CLSID\{6D9B9F18-1DD6-4377-946A-5ED883B78031}\InprocServer32]
@="E:\\COM\\ContainmentOuterComponentWithRegFile\\ContainmentOuterComponentWithRegFile.dll"

[HKEY_CLASSES_ROOT\CLSID\{764E7C57-1737-4f19-927A-6081C39EF514}]
@="InnerComDll_WithRegFile"
[HKEY_CLASSES_ROOT\CLSID\{764E7C57-1737-4f19-927A-6081C39EF514}\InprocServer32]
@="E:\\COM\\ContainmentInnerComponentWithRegFile\\ContainmentInnerComponentWithRegFile.dll"
```

ClientOfContainmentWithRegFile.cpp

```
#include<windows.h>
#include"HeaderForClientOfComponentWithRegFile.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// global variable declarations ISum
*pISum = NULL; ISubtract *pISubtract =
NULL; IMultiplication *pIMultiplication =
NULL; IDivision *pIDivision = NULL;


// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    // variable
    declarations WNDCLASSEX
    wndclass; HWND hwnd;
    MSG msg;
    TCHAR AppName[] =
    TEXT("ComClient"); HRESULT hr;
```

```
    // COM Initialization
    hr = CoInitialize(NULL);

    if (FAILED(hr))
    {
       MessageBox(NULL, TEXT("COM Library Can Not Be Initialized.\nProgram Will Now Exit."),
       TEXT("Program Error"), MB_OK);
       exit(0);
    }

    // WNDCLASSEX initialization wndclass.cbSize =
    sizeof(wndclass); wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.cbClsExtra = 0; wndclass.cbWndExtra = 0;
    wndclass.lpfnWndProc = WndProc; wndclass.hIcon =
    LoadIcon(NULL, IDI_APPLICATION); wndclass.hCursor =
    LoadCursor(NULL, IDC_ARROW); wndclass.hbrBackground =
    (HBRUSH)GetStockObject(WHITE_BRUSH); wndclass.hInstance =
    hInstance;


    wndclass.lpszClassName = AppName;
    wndclass.lpszMenuName = NULL;
    wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // register window class
    RegisterClassEx(&wndclass);

    // create window
    hwnd = CreateWindow(AppName,
       TEXT("Client Of COM Dll Server"),
       WS_OVERLAPPEDWINDOW,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       NULL,
       NULL,
       hInstance,
       NULL);

    ShowWindow(hwnd, nCmdShow);

    UpdateWindow(hwnd);

    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }

    // COM Un-initialization
    CoUninitialize();
    return((int)msg.wParam);
}

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    // function declarations
    void SafeInterfaceRelease(void);

    // variable declarations
    HRESULT hr;
    int iNum1, iNum2, iSum, iSubtraction, iMultiplication,
    iDivision; TCHAR str[255];

    // code
```

```
switch (iMsg)
{
case WM_CREATE:
   hr = CoCreateInstance(CLSID_SumSubtract, NULL,
      CLSCTX_INPROC_SERVER, IID_ISum, (void **)&pISum);

   if (FAILED(hr))
   {
      MessageBox(hwnd, TEXT("ISum Interface Can Not Be Obtained"), TEXT("Error"), MB_OK);
      DestroyWindow(hwnd);
   }

   // initialize arguments
   hardcoded iNum1 = 65;
   iNum2 = 45;

   // call SumOfTwoIntegers() of ISum to get the sum
   pISum->SumOfTwoIntegers(iNum1, iNum2, &iSum);

   // display the result
   wsprintf(str, TEXT("Sum Of %d And %d = %d"), iNum1, iNum2, iSum);
   MessageBox(hwnd, str, TEXT("Result"), MB_OK);

   // call QueryInterface() on ISum,to get ISubtract's pointer
   hr = pISum->QueryInterface(IID_ISubtract, (void **)&pISubtract);

   if (FAILED(hr))
   {
      MessageBox(hwnd, TEXT("ISubtract Interface Can Not Be Obtained"), TEXT("Error"),
      MB_OK);
      DestroyWindow(hwnd);
   }

   // as ISum is now not needed onwords, release
   it pISum->Release();
   pISum = NULL;// make relesed interface NULL
            // again initialize arguments hardcoded
   iNum1 = 155;
   iNum2 = 55;
   // call SubtractionOfTwoIntegers() of ISubtract to get the subtraction
   pISubtract->SubtractionOfTwoIntegers(iNum1, iNum2, &iSubtraction);

   // display the result
   wsprintf(str, TEXT("Subtraction Of %d And %d = %d"), iNum1, iNum2, iSubtraction);
   MessageBox(hwnd, str, TEXT("Result"), MB_OK);

   // call QueryInterface() on ISubtract,to get IMultiplication's pointer
   hr = pISubtract->QueryInterface(IID_IMultiplication, (void **)&pIMultiplication);

   if (FAILED(hr))
   {
      MessageBox(hwnd, TEXT("IMultiplication Interface Can Not Be Obtained"), TEXT("Error"),
      MB_OK);
      DestroyWindow(hwnd);
   }

   // as ISubtract is now not needed onwords, release
   it pISubtract->Release();
   pISubtract = NULL;// make relesed interface NULL
                // again initialize arguments hardcoded
   iNum1 = 30;
   iNum2 = 25;
   // call MultiplicationOfTwoIntegers() of IMultiplication to get the Multiplication
   pIMultiplication->MultiplicationOfTwoIntegers(iNum1, iNum2, &iMultiplication);



   // display the result
```

```
        wsprintf(str, TEXT("Multiplication Of %d And %d = %d"), iNum1, iNum2, iMultiplication);

        MessageBox(hwnd, str, TEXT("Result"), MB_OK);

        // call QueryInterface() on IMultiplication's to get IDivision pointer
        hr = pIMultiplication->QueryInterface(IID_IDivision, (void **)&pIDivision);

        if (FAILED(hr))
        {
           MessageBox(hwnd, TEXT("IDivision Interface Can Not Be Obtained"), TEXT("Error"),
           MB_OK);
           DestroyWindow(hwnd);
        }

        // as IMultiplication is now not needed onwords, release it
        pIMultiplication->Release();
        pIMultiplication = NULL;// make relesed interface NULL
                         // again initialize arguments hardcoded
        iNum1 = 200;
        iNum2 = 25;

        // call DivisionOfTwoIntegers() of IDivision to get the Division
        pIDivision->DivisionOfTwoIntegers(iNum1, iNum2, &iDivision);

        // display the result
        wsprintf(str, TEXT("Division Of %d And %d = %d"), iNum1, iNum2,
        iDivision); MessageBox(hwnd, str, TEXT("Result"), MB_OK);

        // finally release IDivision
        pIDivision->Release();
        pIDivision = NULL;// make relesed interface NULL
                      // exit the
        application DestroyWindow(hwnd);
        break;
    case WM_DESTROY:
        SafeInterfaceRelease();
        PostQuitMessage(0);
        break;
    }
    return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}

void SafeInterfaceRelease(void)
{
    if (pISum)
    {
        pISum->Release();
        pISum = NULL;
    }

    if (pISubtract)
    {
        pISubtract->Release();
        pISubtract = NULL;
    }

    if (pIMultiplication)
    {
        pIMultiplication->Release();
        pIMultiplication = NULL;
    }

    if (pIDivision)
    {
        pIDivision->Release();
        pIDivision = NULL;
    }
}
```

4.2 Aggregation

The outer object wishes to expose interfaces from the inner object as if they were implemented on the outer object itself. This is useful when the outer object would always delegate every call to one of its interfaces to the same interface of the inner object. Aggregation is a convenience to allow the outer object to avoid extra implementation overhead in such cases.

In Aggregation, you need to implement three IUnknown functions: QueryInterface, AddRef, and Release. The catch is that from the client's perspective, any IUnknown function on the outer object must affect the outer object. That is, AddRef and Release affect the outer object and QueryInterface exposes all the interfaces available on the outer object. However, if the outer object simply exposes an inner object's interface as its own, that inner object's IUnknown members called through that interface will behave differently than those IUnknown members on the outer object's interfaces, a sheer violation of the rules and properties governing IUnknown.

The solution is for the outer object to somehow pass the inner object some IUnknown pointer to which the inner object can re-route (that is, delegate) IUnknown calls in its own interfaces, and yet there must be a method through which the outer object can access the inner object's IUnknown functions that only affect the inner object.

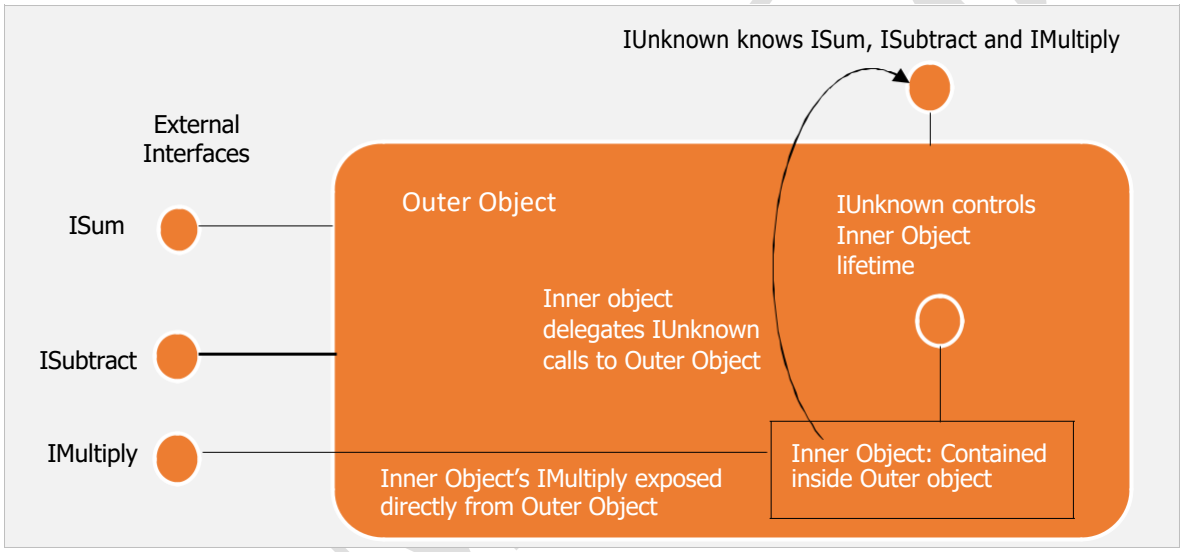


Figure 4.2: Aggregation of an inner object where the outer object exposes one or more of the inner object's interface as its own.

4.2.1 Server/Client Implementation

Program structure:

Aggregation Server:
Inner component:

1. AggregationInnerComponentWithRegFile.h – to declare public interface consumable by clients (inner component exposed interfaces)
2. AggregationInnerComponentWithRegFile.cpp – to implement/define public and private operations for inner component
3. AggregationInnerComponentWithRegFile.def – inner component's exported methods

Outer component:

1. AggregationOuterComponentWithRegFile.h – to declare public interface consumable by clients (outer component exposed interfaces)
2. AggregationInnerComponentWithRegFile.h – to declare public interface consumable by clients (inner component exposed interfaces)
3. AggregationOuterComponentWithRegFile.cpp – to implement/define public and private operations for outer component
4. AggregationOuterComponentWithRegFile.def – outer component's exported methods

Aggregation Client –

1. HeaderForClientOfComponentWithRegFile.h – DLL server's header file to get public interfaces (inner and outer components' merged header)
2. ClientOfAggregationWithRegFile.cpp – client implementation to consume inner/outer component
3. RegisterServer.reg – registry file to register inner and outer components into the system registry

4.2.1.1 Aggregation Inner Component

AggregationInnerComponentWithRegFile.h

```
class IMultiplication :public IUnknown
{
public:
   // IMultiplication specific method declarations pure virtual
   virtual HRESULT __stdcall MultiplicationOfTwoIntegers(int, int, int *) = 0;
};
class IDivision :public IUnknown
{
public:
   // IDivision specific method declarations
   virtual HRESULT __stdcall DivisionOfTwoIntegers(int, int, int *) = 0;// pure virtual
};

// CLSID of MultiplicationDivision Component {4AC37EEC-DAFD-435d-A1BC-261EFA28EF46}
const CLSID CLSID_MultiplicationDivision = {
0x4ac37eec,0xdafd,0x435d,0xa1,0xbc,0x26,0x1e,0xfa,0x28,0xef,0x46 };

// IID of IMultiplication Interface
const IID IID_IMultiplication = {
0xa39f8306,0x7a0c,0x4e47,0xb3,0x8a,0xfc,0x8d,0x68,0x5d,0xca,0x90 };

// IID of IDivision Interface
const IID IID_IDivision = { 0x9f7d9e5a,0xab6,0x4a59,0xad,0x22,0xa,0x89,0x88,0x2e,0x46,0x28 };
```

AggregationInnerComponentWithRegFile.def

```
LIBRARY AggregationInnerComponentWithRegFile
EXPORTS
      DllGetClassObject      PRIVATE
      DllCanUnloadNow     PRIVATE
```

AggregationInnerComponentWithRegFile.cpp

```
#include<windows.h>
#include"AggregationInnerComponentWithRegFile.h"

// interface declaration ( for internal use only. i.e. not to be included in .h file )
interface INoAggregationIUnknown // new
{
   virtual HRESULT __stdcall QueryInterface_NoAggregation(REFIID, void **) =
   0; virtual ULONG __stdcall AddRef_NoAggregation(void) = 0;
   virtual ULONG __stdcall Release_NoAggregation(void) = 0;
};

// class declarations
class CMultiplicationDivision :public INoAggregationIUnknown, IMultiplication, IDivision
{
private:
   long m_cRef;
   IUnknown *m_pIUnknownOuter;
public:
   CMultiplicationDivision(IUnknown *); // constructor method declarations

   ~CMultiplicationDivision(void); // destructor method declarations

   // Aggregation Supported IUnknown specific method declarations (inherited)
```

```
    HRESULT __stdcall QueryInterface(REFIID, void **);
    ULONG __stdcall AddRef(void);
    ULONG __stdcall Release(void);

    // Aggregation NonSupported IUnknown specific method declarations (inherited)
    HRESULT __stdcall QueryInterface_NoAggregation(REFIID, void **);// new ULONG
    __stdcall AddRef_NoAggregation(void);// new
    ULONG __stdcall Release_NoAggregation(void);// new

        // IMultiplication specific method declarations (inherited)
    HRESULT __stdcall MultiplicationOfTwoIntegers(int, int, int *);

    // IDivision specific method declarations (inherited)
    HRESULT __stdcall DivisionOfTwoIntegers(int, int, int *);
};
class CMultiplicationDivisionClassFactory :public IClassFactory
{
private:
    long m_cRef;
public:
    // constructor method declarations
    CMultiplicationDivisionClassFactory(void);
    // destructor method declarations
    ~CMultiplicationDivisionClassFactory(void);

    // IUnknown specific method declarations (inherited)
    HRESULT __stdcall QueryInterface(REFIID, void **);
    ULONG __stdcall AddRef(void);
    ULONG __stdcall Release(void);

    // IClassFactory specific method declarations (inherited)
    HRESULT __stdcall CreateInstance(IUnknown *, REFIID, void
    **); HRESULT __stdcall LockServer(BOOL);
};

// global variable declarations
long glNumberOfActiveComponents = 0;// number of active components
long glNumberOfServerLocks = 0;// number of locks on this dll
                        // DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
    // code
    switch (dwReason)
    {
          case DLL_PROCESS_ATTACH:
             break;
          case DLL_PROCESS_DETACH:
             break;
    }

    return(TRUE);
}

// Implementation Of CMultiplicationDivision's Constructor Method
CMultiplicationDivision::CMultiplicationDivision(IUnknown *pIUnknownOuter)
{
    m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()

    InterlockedIncrement(&glNumberOfActiveComponents);// increment global counter

    if (pIUnknownOuter != NULL)
       m_pIUnknownOuter = pIUnknownOuter;
    else
       m_pIUnknownOuter = reinterpret_cast<IUnknown *>
                        (static_cast<INoAggregationIUnknown *>(this));
}
```

```
// Implementation Of CSumSubtract's Destructor Method
CMultiplicationDivision::~CMultiplicationDivision(void)
{
   InterlockedDecrement(&glNumberOfActiveComponents);// decrement global counter
}

// Implementation Of CMultiplicationDivision's Aggrgation Supporting IUnknown's Methods
HRESULT CMultiplicationDivision::QueryInterface(REFIID riid, void **ppv)
{
   return(m_pIUnknownOuter->QueryInterface(riid, ppv));
}



ULONG CMultiplicationDivision::AddRef(void)
{
   return(m_pIUnknownOuter->AddRef());
}

ULONG CMultiplicationDivision::Release(void)
{
   return(m_pIUnknownOuter->Release());
}

// Implementation Of CMultiplicationDivision's Aggregation NonSupporting IUnknown's Methods
HRESULT CMultiplicationDivision::QueryInterface_NoAggregation(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<INoAggregationIUnknown
   *>(this); else if (riid == IID_IMultiplication)
      *ppv = static_cast<IMultiplication
   *>(this); else if (riid == IID_IDivision)
      *ppv = static_cast<IDivision
   *>(this); else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

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

ULONG CMultiplicationDivision::AddRef_NoAggregation(void)
{
   InterlockedIncrement(&m_cRef);
   return(m_cRef);
}

ULONG CMultiplicationDivision::Release_NoAggregation(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }
   return(m_cRef);
}

// Implementation Of IMultiplication's Methods
HRESULT CMultiplicationDivision::MultiplicationOfTwoIntegers(int num1, int num2, int
*pMultiplication)
{
   *pMultiplication = num1*num2;
   return(S_OK);
}
```

```
// Implementation Of IDivision's Methods
HRESULT CMultiplicationDivision::DivisionOfTwoIntegers(int num1, int num2, int *pDivision)
{
    *pDivision = num1 / num2;
    return(S_OK);
}

// Implementation Of CMultiplicationDivisionClassFactory's Constructor Method
CMultiplicationDivisionClassFactory::CMultiplicationDivisionClassFactory(void)
{
    m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()
}

// Implementation Of CMultiplicationDivisionClassFactory's Destructor Method
CMultiplicationDivisionClassFactory::~CMultiplicationDivisionClassFactory(void)
{
    // no code
}

// Implementation Of CMultiplicationDivisionClassFactory's IClassFactory's IUnknown's Methods
HRESULT CMultiplicationDivisionClassFactory::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IClassFactory *>(this);
    else if (riid == IID_IClassFactory)
        *ppv = static_cast<IClassFactory
    *>(this); else
    {
        *ppv = NULL;
        return(E_NOINTERFACE);
    }

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

ULONG CMultiplicationDivisionClassFactory::AddRef(void)
{
    InterlockedIncrement(&m_cRef);
    return(m_cRef);
}

ULONG CMultiplicationDivisionClassFactory::Release(void)
{
    InterlockedDecrement(&m_cRef);

    if (m_cRef == 0)
    {
        delete(this);
        return(0);
    }

    return(m_cRef);
}

// Implementation Of CMultiplicationDivisionClassFactory's IClassFactory's Methods
HRESULT CMultiplicationDivisionClassFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid,
void **ppv)
{
    // variable declarations
    CMultiplicationDivision *pCMultiplicationDivision =
    NULL; HRESULT hr;

    if ((pUnkOuter != NULL) && (riid != IID_IUnknown))
        return(CLASS_E_NOAGGREGATION);

    // create the instance of component i.e. of CMultiplicationDivision
    class pCMultiplicationDivision = new CMultiplicationDivision(pUnkOuter);
```

```
    if (pCMultiplicationDivision == NULL)
        return(E_OUTOFMEMORY);

    // get the requested interface
    hr = pCMultiplicationDivision->QueryInterface_NoAggregation(riid, ppv);

        // anticipate possible failure of QueryInterface()
    pCMultiplicationDivision->Release_NoAggregation();

    return(hr);
}

HRESULT CMultiplicationDivisionClassFactory::LockServer(BOOL fLock)
{
    if (fLock)
        InterlockedIncrement(&glNumberOfServerLocks);
    else
        InterlockedDecrement(&glNumberOfServerLocks);

    return(S_OK);
}

// Implementation Of Exported Functions From This Dll
HRESULT __stdcall DllGetClassObject(REFCLSID rclsid, REFIID riid, void **ppv)
{
    // variable declaraions
    CMultiplicationDivisionClassFactory *pCMultiplicationDivisionClassFactory =
    NULL; HRESULT hr;

    if (rclsid != CLSID_MultiplicationDivision)
        return(CLASS_E_CLASSNOTAVAILABLE);

    // create class factory
    pCMultiplicationDivisionClassFactory = new CMultiplicationDivisionClassFactory;

    if (pCMultiplicationDivisionClassFactory == NULL)
        return(E_OUTOFMEMORY);

    hr = pCMultiplicationDivisionClassFactory->QueryInterface(riid, ppv);

        // anticipate possible failure of
    QueryInterface() pCMultiplicationDivisionClassFactory-
    >Release(); return(hr);
}

HRESULT __stdcall DllCanUnloadNow(void)
{
    if ((glNumberOfActiveComponents == 0) && (glNumberOfServerLocks == 0))
        return(S_OK);
    else
        return(S_FALSE);
}
```

4.2.1.2 Aggregation Outer Component

AggregationInnerComponentWithRegFile.h – refer to inner's header file

AggregationOuterComponentWithRegFile.h

```
class ISum :public IUnknown
{
public:
    // ISum specific method declarations
    virtual HRESULT __stdcall SumOfTwoIntegers(int, int, int *) = 0;// pure virtual
};
```

```
class ISubtract :public IUnknown
{
public:
   // ISubtract specific method declarations
   virtual HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *) = 0;// pure virtual
};

// CLSID of SumSubtract Component {FA5EC586-38C0-4d05-9744-A52D4B13292A}
const CLSID CLSID_SumSubtract = { 0xfa5ec586, 0x38c0, 0x4d05, 0x97, 0x44, 0xa5, 0x2d, 0x4b,
0x13, 0x29, 0x2a };

// IID of ISum Interface
const IID IID_ISum = { 0x791876b8,0x4bd,0x4202,0x91,0x8d,0xc2,0x66,0x30,0x96,0xfe,0xbf };

// IID of ISubtract Interface
const IID IID_ISubtract = { 0x9f2a8316,0x4eda,0x4113,0xac,0x4c,0x64,0x52,0x24,0x18,0x78,0x47 };
```

AggregationOuterComponentWithRegFile.def

```
LIBRARY AggregationOuterComponentWithRegFile
EXPORTS
      DllGetClassObject    PRIVATE
      DllCanUnloadNow      PRIVATE
```

AggregationOuterComponentWithRegFile.cpp

```
#include<windows.h>
#include"AggregationInnerComponentWithRegFile.h"
#include"AggregationOuterComponentWithRegFile.h"

// class declarations
class CSumSubtract :public ISum, ISubtract
{
private:
   long m_cRef;
   IUnknown *m_pIUnknownInner;
   IMultiplication *m_pIMultiplication;
   IDivision *m_pIDivision;
public:
   // constructor method
   declarations CSumSubtract(void);
   // destructor method declarations
   ~CSumSubtract(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // ISum specific method declarations (inherited)
   HRESULT __stdcall SumOfTwoIntegers(int, int, int *);

   // ISubtract specific method declarations (inherited) HRESULT
   __stdcall SubtractionOfTwoIntegers(int, int, int *);

   // custom method for inner component creation
   HRESULT __stdcall InitializeInnerComponent(void);
};

class CSumSubtractClassFactory :public IClassFactory
{
private:
   long m_cRef;
public:
   // constructor method declarations
   CSumSubtractClassFactory(void);
```

```
    // destructor method declarations
    ~CSumSubtractClassFactory(void);

    // IUnknown specific method declarations (inherited)
    HRESULT __stdcall QueryInterface(REFIID, void **);
    ULONG __stdcall AddRef(void);
    ULONG __stdcall Release(void);

    // IClassFactory specific method declarations (inherited) HRESULT
    __stdcall CreateInstance(IUnknown *, REFIID, void **);

    HRESULT __stdcall LockServer(BOOL);
};

// global variable declarations
long glNumberOfActiveComponents = 0;// number of active components
long glNumberOfServerLocks = 0;// number of locks on this dll
                        // DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
    switch (dwReason)
    {
          case DLL_PROCESS_ATTACH:
             break;
          case DLL_PROCESS_DETACH:
             break;
    }

    return(TRUE);
}

// Implementation Of CSumSubtract's Constructor Method
CSumSubtract::CSumSubtract(void)
{
    // initialization of private data members
    m_pIUnknownInner = NULL;
    m_pIMultiplication = NULL;
    m_pIDivision = NULL;
    m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()

    InterlockedIncrement(&glNumberOfActiveComponents);// increment global counter
}

// Implementation Of CSumSubtract's Destructor Method
CSumSubtract::~CSumSubtract(void)
{
    InterlockedDecrement(&glNumberOfActiveComponents);// decrement global counter

    if (m_pIMultiplication)
    {
       m_pIMultiplication->Release();
       m_pIMultiplication = NULL;
    }

    if (m_pIDivision)
    {
       m_pIDivision->Release();
       m_pIDivision = NULL;
    }

    if (m_pIUnknownInner)
    {
       m_pIUnknownInner->Release();
       m_pIUnknownInner = NULL;
    }
}
```

```
// Implementation Of CSumSubtract's IUnknown's Methods
HRESULT CSumSubtract::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<ISum *>(this);
    else if (riid == IID_ISum)
        *ppv = static_cast<ISum *>(this);
    else if (riid == IID_ISubtract)
        *ppv = static_cast<ISubtract *>(this);
    else if (riid == IID_IMultiplication)
        return(m_pIUnknownInner->QueryInterface(riid,
    ppv)); else if (riid == IID_IDivision)
        return(m_pIUnknownInner->QueryInterface(riid,
    ppv)); else
    {
        *ppv = NULL;
        return(E_NOINTERFACE);
    }

    reinterpret_cast<IUnknown *>(*ppv)->AddRef();

    return(S_OK);
}

ULONG CSumSubtract::AddRef(void)
{
    InterlockedIncrement(&m_cRef);
    return(m_cRef);
}

ULONG CSumSubtract::Release(void)
{
    InterlockedDecrement(&m_cRef);

    if (m_cRef == 0)
    {
        delete(this);
        return(0);
    }

    return(m_cRef);
}

// Implementation Of ISum's Methods
HRESULT CSumSubtract::SumOfTwoIntegers(int num1, int num2, int *pSum)
{
    *pSum = num1 + num2;
    return(S_OK);
}

// Implementation Of ISubtract's Methods
HRESULT CSumSubtract::SubtractionOfTwoIntegers(int num1, int num2, int *pSubtract)
{
    *pSubtract = num1 - num2;
    return(S_OK);
}
HRESULT CSumSubtract::InitializeInnerComponent(void)
{
    // variable declarations
    HRESULT hr;

    hr = CoCreateInstance(CLSID_MultiplicationDivision,
        reinterpret_cast<IUnknown *>(this),
        CLSCTX_INPROC_SERVER,
        IID_IUnknown,
        (void **)&m_pIUnknownInner);

    if (FAILED(hr))
```

```
    {
        MessageBox(NULL, TEXT("IUnknown Interface Can Not Be Obtained From
        Inner Component."), TEXT("Error"), MB_OK);

        return(E_FAIL);
    }

    hr = m_pIUnknownInner->QueryInterface(IID_IMultiplication, (void **)
    &m_pIMultiplication);

    if (FAILED(hr))
    {
        MessageBox(NULL, TEXT("IMultiplication Interface Can Not Be Obtained From Inner
        Component."), TEXT("Error"), MB_OK);

        m_pIUnknownInner->Release();
        m_pIUnknownInner = NULL;

        return(E_FAIL);
    }

    hr = m_pIUnknownInner->QueryInterface(IID_IDivision, (void **)&m_pIDivision);

    if (FAILED(hr))
    {
        MessageBox(NULL, TEXT("IDivision Interface Can Not Be Obtained From
        Inner Component."), TEXT("Error"), MB_OK);

        m_pIUnknownInner->Release();
        m_pIUnknownInner = NULL;

        return(E_FAIL);
    }

    return(S_OK);
}

// Implementation Of CSumSubtractClassFactory's Constructor Method
CSumSubtractClassFactory::CSumSubtractClassFactory(void)
{
    // hardcoded initialization to anticipate possible failure of
    QueryInterface() m_cRef = 1;
}

// Implementation Of CSumSubtractClassFactory's Destructor Method
CSumSubtractClassFactory::~CSumSubtractClassFactory(void)
{
    // no code
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's IUnknown's Methods
HRESULT CSumSubtractClassFactory::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IClassFactory *>(this);
    else if (riid == IID_IClassFactory)
        *ppv = static_cast<IClassFactory
    *>(this); else
    {
        *ppv = NULL;
        return(E_NOINTERFACE);
    }

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

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

ULONG CSumSubtractClassFactory::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's Methods
HRESULT CSumSubtractClassFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid, void **ppv)
{
   // variable declarations
   CSumSubtract *pCSumSubtract =
   NULL; HRESULT hr;

   if (pUnkOuter != NULL)
      return(CLASS_E_NOAGGREGATION);

   // create the instance of component i.e. of CSumSubtract
   class pCSumSubtract = new CSumSubtract;

   if (pCSumSubtract == NULL)
      return(E_OUTOFMEMORY);

   // initialize the inner component
   hr = pCSumSubtract->InitializeInnerComponent();

   if (FAILED(hr))
   {
               MessageBox(NULL, TEXT("Failed To Initialize Inner Component"), TEXT("Error"),
               MB_OK);
      pCSumSubtract->Release();
      return(hr);
   }

   // get the requested interface
   hr = pCSumSubtract->QueryInterface(riid, ppv);

   pCSumSubtract->Release();// anticipate possible failure of QueryInterface()
   return(hr);
}

HRESULT CSumSubtractClassFactory::LockServer(BOOL fLock)
{
   if (fLock)
      InterlockedIncrement(&glNumberOfServerLocks);
   else
      InterlockedDecrement(&glNumberOfServerLocks);

   return(S_OK);
}

// Implementation Of Exported Functions From This Dll
HRESULT __stdcall DllGetClassObject(REFCLSID rclsid, REFIID riid, void **ppv)
{
   // variable declaraions
   CSumSubtractClassFactory *pCSumSubtractClassFactory = NULL;
```

```
    HRESULT hr;

    if (rclsid != CLSID_SumSubtract)
        return(CLASS_E_CLASSNOTAVAILABLE);

    // create class factory
    pCSumSubtractClassFactory = new CSumSubtractClassFactory;

    if (pCSumSubtractClassFactory == NULL)
        return(E_OUTOFMEMORY);

    hr = pCSumSubtractClassFactory->QueryInterface(riid, ppv); pCSumSubtractClassFactory-
    >Release();// anticipate possible failure of QueryInterface()

    return(hr);
}

HRESULT __stdcall DllCanUnloadNow(void)
{
    if ((glNumberOfActiveComponents == 0) && (glNumberOfServerLocks == 0))
        return(S_OK);
    else
        return(S_FALSE);
}
```

4.2.1.3 Aggregation Client

HeaderForClientOfComponentWithRegFile.h – Merge inner and outer components' header files together
ClientOfAggregationWithRegFile.cpp

```
#include<windows.h>
#include"HeaderForClientOfComponentWithRegFile.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// global variable declarations ISum
*pISum = NULL; ISubtract *pISubtract =
NULL; IMultiplication *pIMultiplication =
NULL; IDivision *pIDivision = NULL;


// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    // variable
    declarations WNDCLASSEX
    wndclass; HWND hwnd;
    MSG msg;
    TCHAR AppName[] = TEXT("ComClient");
        HRESULT hr;


    // code
    // COM Initialization
    hr = CoInitialize(NULL);

    if (FAILED(hr))
    {
                MessageBox(NULL, TEXT("COM Library Can Not Be Initialized.\nProgram Will Now
                Exit."), TEXT("Program Error"), MB_OK);
        exit(0);
    }

    // WNDCLASSEX initialization
    wndclass.cbSize = sizeof(wndclass);
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
```

```
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.lpfnWndProc = WndProc;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.hInstance = hInstance;
    wndclass.lpszClassName = AppName;
    wndclass.lpszMenuName = NULL;
    wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // register window class
    RegisterClassEx(&wndclass);

    // create window
    hwnd = CreateWindow(AppName,
       TEXT("Client Of COM Dll Server"),
       WS_OVERLAPPEDWINDOW,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       NULL,
       NULL,
       hInstance,
       NULL);

    ShowWindow(hwnd, nCmdShow);

    UpdateWindow(hwnd);

    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }

    // COM Un-initialization
    CoUninitialize();

    return((int)msg.wParam);
}

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    // function declarations
    void SafeInterfaceRelease(void);

    // variable declarations
    HRESULT hr;
    int iNum1, iNum2, iSum, iSubtraction, iMultiplication,
    iDivision; TCHAR str[255];

    // code
    switch (iMsg)
    {
    case WM_CREATE:
       hr = CoCreateInstance(CLSID_SumSubtract, NULL,
          CLSCTX_INPROC_SERVER, IID_ISum, (void **)&pISum);

       if (FAILED(hr))
       {
                      MessageBox(hwnd, TEXT("ISum Interface Can Not Be Obtained"),
                      TEXT("Error"), MB_OK);
          DestroyWindow(hwnd);
       }
```

```
// initialize arguments
hardcoded iNum1 = 65;
iNum2 = 45;

// call SumOfTwoIntegers() of ISum to get the sum
pISum->SumOfTwoIntegers(iNum1, iNum2, &iSum);

// display the result
wsprintf(str, TEXT("Sum Of %d And %d = %d"), iNum1, iNum2, iSum);
MessageBox(hwnd, str, TEXT("Result"), MB_OK);

// call QueryInterface() on ISum,to get ISubtract's pointer
hr = pISum->QueryInterface(IID_ISubtract, (void **)&pISubtract);

if (FAILED(hr))
{
                MessageBox(hwnd, TEXT("ISubtract Interface Can Not Be Obtained"),
                TEXT("Error"), MB_OK);
   DestroyWindow(hwnd);
}

// as ISum is now not needed onwords, release
it pISum->Release();
pISum = NULL;// make relesed interface NULL
          // again initialize arguments hardcoded
iNum1 = 155;
iNum2 = 55;
// call SubtractionOfTwoIntegers() of ISubtract to get the subtraction
pISubtract->SubtractionOfTwoIntegers(iNum1, iNum2, &iSubtraction);

// display the result
wsprintf(str, TEXT("Subtraction Of %d And %d = %d"), iNum1, iNum2, iSubtraction);
MessageBox(hwnd, str, TEXT("Result"), MB_OK);

// call QueryInterface() on ISubtract,to get IMultiplication's pointer
hr = pISubtract->QueryInterface(IID_IMultiplication, (void **)&pIMultiplication);

if (FAILED(hr))
{
                MessageBox(hwnd, TEXT("IMultiplication Interface Can Not Be Obtained"),
                TEXT("Error"), MB_OK);
   DestroyWindow(hwnd);
}

// as ISubtract is now not needed onwords, release
it pISubtract->Release();
pISubtract = NULL;// make relesed interface NULL
            // again initialize arguments hardcoded
iNum1 = 30;
iNum2 = 25;
// call MultiplicationOfTwoIntegers() of IMultiplication to get the Multiplication
pIMultiplication->MultiplicationOfTwoIntegers(iNum1, iNum2, &iMultiplication);

// display the result
        wsprintf(str, TEXT("Multiplication Of %d And %d = %d"), iNum1, iNum2,
        iMultiplication);
MessageBox(hwnd, str, TEXT("Result"), MB_OK);

// call QueryInterface() on IMultiplication's to get IDivision pointer
hr = pIMultiplication->QueryInterface(IID_IDivision, (void **)&pIDivision);

if (FAILED(hr))
{
                MessageBox(hwnd, TEXT("IDivision Interface Can Not Be Obtained"),
                TEXT("Error"), MB_OK);
   DestroyWindow(hwnd);
}
```

```
        // as IMultiplication is now not needed onwords, release it
        pIMultiplication->Release();
        pIMultiplication = NULL;// make relesed interface NULL
                        // again initialize arguments hardcoded
        iNum1 = 200;
        iNum2 = 25;

        // call DivisionOfTwoIntegers() of IDivision to get the Division
        pIDivision->DivisionOfTwoIntegers(iNum1, iNum2, &iDivision);

        // display the result
        wsprintf(str, TEXT("Division Of %d And %d = %d"), iNum1, iNum2,
        iDivision); MessageBox(hwnd, str, TEXT("Result"), MB_OK);

        // finally release IDivision
        pIDivision->Release();
        pIDivision = NULL;// make relesed interface NULL
                    // exit the
        application DestroyWindow(hwnd);

        break;
    case WM_DESTROY:
        SafeInterfaceRelease();
        PostQuitMessage(0);

        break;
    }

    return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}

void SafeInterfaceRelease(void)
{
    // code
    if (pISum)
    {
        pISum->Release();
        pISum = NULL;
    }

    if (pISubtract)
    {
        pISubtract->Release();
        pISubtract = NULL;
    }

    if (pIMultiplication)
    {
        pIMultiplication->Release();
        pIMultiplication = NULL;
    }

    if (pIDivision)
    {
        pIDivision->Release();
        pIDivision = NULL;
    }
}
```

Page intentionally left blank

Page intentionally left blank

5. EXE (out-of-process) server

One of the most important ways for a client to get a pointer to an object is for the client to ask that a server be launched and that an instance of the object provided by the server be created and activated. It is the responsibility of the server to ensure that this happens properly.

The server must implement code for a class object through an implementation of either the IClassFactory or IClassFactory2 interface.

The server must register its CLSID in the system registry on the machine on which it resides and further, has the option of publishing its machine location to other systems on a network to allow clients to call it without requiring the client to know the server's location.

To create the out-of-proc COM server in C++, we'll need to essentially create an in-proc server, but with a couple of extra steps to deal with moving data across process boundaries, and with some changes to other steps.

5.1. Server implementation

Program structure:

1. ExeServerWithRegFile.h - to declare public interface consumable by clients
2. ExeServerWithRegFile.cpp - to implement/define public and private operations
3. ExeServer.reg - registry file to register inner and outer components into the system registry

ExeServerWithRegFile.h

```
#pragma once

class ISum :public IUnknown
{
public:
   // ISum specific method declarations
   virtual HRESULT __stdcall SumOfTwoIntegers(int, int, int *) = 0;// pure virtual
};

class ISubtract :public IUnknown
{
public:
   // ISubtract specific method declarations
   virtual HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *) = 0;// pure virtual
};

// CLSID of SumSubtract Component {7ACCABF1-40CE-49a5-8703-0183CDE6B91F}
const CLSID CLSID_SumSubtract = {
0x7accabf1,0x40ce,0x49a5,0x87,0x3,0x1,0x83,0xcd,0xe6,0xb9,0x1f };

// IID of ISum Interface {8CC14612-CAB1-42dc-8E22-B88F5B2F0CC2}
const IID IID_ISum = { 0x8cc14612,0xcab1,0x42dc,0x8e,0x22,0xb8,0x8f,0x5b,0x2f,0xc,0xc2 };

// IID of ISubtract Interface {B16D9F59-BFA4-4c2d-90C6-7AC06B08BE91}
const IID IID_ISubtract = { 0xb16d9f59,0xbfa4,0x4c2d,0x90,0xc6,0x7a,0xc0,0x6b,0x8,0xbe,0x91 };
```

ExeServerWithRegFile.cpp

```
#define UNICODE

#include<windows.h>
#include<tlhelp32.h> // for process snapshot related apis and
structures #include"ExeServerWithRegFile.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// class declarations
class CSumSubtract :public ISum, ISubtract
```

```
{
private:
   long m_cRef;
public:
   // constructor method
   declarations CSumSubtract(void);
   // destructor method declarations
   ~CSumSubtract(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // ISum specific method declarations (inherited)
   HRESULT __stdcall SumOfTwoIntegers(int, int, int *);

   // ISubtract specific method declarations (inherited) HRESULT
   __stdcall SubtractionOfTwoIntegers(int, int, int *);
};

class CSumSubtractClassFactory :public IClassFactory
{
private:
   long m_cRef;
public:
   // constructor method declarations
   CSumSubtractClassFactory(void);
   // destructor method declarations
   ~CSumSubtractClassFactory(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // IClassFactory specific method declarations (inherited)
   HRESULT __stdcall CreateInstance(IUnknown *, REFIID, void
   **); HRESULT __stdcall LockServer(BOOL);
};

// global variable declarations
long glNumberOfActiveComponents = 0; // number of active components
long glNumberOfServerLocks = 0; // number of locks on this dll
IClassFactory *gpIClassFactory = NULL;
HWND ghwnd = NULL;
DWORD dwRegister;

// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
   // function declarations
   HRESULT StartMyClassFactories(void);
   void StopMyClassFactories(void);
   DWORD GetParentProcessID(void);

   // variable
   declarations WNDCLASSEX
   wndclass; MSG msg;
   HWND hwnd;
   HRESULT hr;
   int iDontShowWindow = 0;// 0 means show the
   window TCHAR AppName[] = TEXT("ExeServer");
   TCHAR szTokens[] = TEXT("-
   /"); TCHAR *pszTokens;
   TCHAR lpszCmdLine[255];
   wchar_t *next_token = NULL;
```

```
// com library initialization
GetParentProcessID();

hr = CoInitialize(NULL);

if (FAILED(hr))
   return(0);

MultiByteToWideChar(CP_ACP, 0, lpCmdLine, 255, lpszCmdLine, 255);

pszTokens = wcstok_s(lpszCmdLine, szTokens, &next_token);

while (pszTokens != NULL)
{
   if (_wcsicmp(pszTokens, TEXT("Embedding")) == 0)// i.e. COM is calling me
   {
      iDontShowWindow = 1; // dont show window but message loop
      must break;
   }
   else
   {
                  MessageBox(NULL, TEXT("Bad Command Line Arguments.\nExitting The
                  Application."), TEXT("Error"), MB_OK);
      exit(0);
   }

   pszTokens = wcstok_s(NULL, szTokens, &next_token);
}

// window code
wndclass.cbSize = sizeof(wndclass);
wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.lpfnWndProc = WndProc;
wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
wndclass.hInstance = hInstance;
wndclass.lpszClassName = AppName;
wndclass.lpszMenuName = NULL;
wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

// register window class
RegisterClassEx(&wndclass);

// create window
hwnd = CreateWindow(AppName,
   TEXT("Exe Server With Reg File"),
   WS_OVERLAPPEDWINDOW,
   CW_USEDEFAULT,
   CW_USEDEFAULT,
   CW_USEDEFAULT,
   CW_USEDEFAULT,
   NULL,
   NULL,
   hInstance,
   NULL);

// initialize global window
handle ghwnd = hwnd;

if (iDontShowWindow != 1) // true if server is not called by COM engine
{
   ShowWindow(hwnd, nCmdShow);
   UpdateWindow(hwnd);

   // increament server lock
```

```
        ++glNumberOfServerLocks;
    }

    if (iDontShowWindow == 1)// only when COM calls this program
    {
        // start class factory
        hr = StartMyClassFactories();

        if (FAILED(hr))
        {
            DestroyWindow(hwnd);
            exit(0);
        }
    }

    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    if (iDontShowWindow == 1)// only when COM calls this program
    {
        // stop class factory
        StopMyClassFactories();
    }

    // com library un-initialization
    CoUninitialize();

    return((int)msg.wParam);
}

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    // variable
    declarations HDC hdc;
    RECT rc;
    PAINTSTRUCT ps;

    switch (iMsg)
    {
          case WM_PAINT:
             GetClientRect(hwnd, &rc);
             hdc = BeginPaint(hwnd, &ps);
             SetBkColor(hdc, RGB(0, 0, 0));
             SetTextColor(hdc, RGB(0, 255, 0));
             DrawText(hdc, TEXT("This Is A COM Exe Server Program. Not For You!!!"),
                 -1, &rc, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
             EndPaint(hwnd, &ps);
             break;
          case WM_DESTROY:
             if (glNumberOfActiveComponents == 0 && glNumberOfServerLocks == 0)
                PostQuitMessage(0);
             break;
          case WM_CLOSE:
             --glNumberOfServerLocks;
             ShowWindow(hwnd, SW_HIDE);
             // fall through, hence no break
          default:
             return(DefWindowProc(hwnd, iMsg, wParam, lParam));
    }

    return(0L);
}
```

```
// Implementation Of CSumSubtract's Constructor Method
CSumSubtract::CSumSubtract(void)
{
   m_cRef = 1;// hardcoded initialization to anticipate possible failure of QueryInterface()
   InterlockedIncrement(&glNumberOfActiveComponents);// increment global counter
}

// Implementation Of CSumSubtract's Destructor Method
CSumSubtract::~CSumSubtract(void)
{
   InterlockedDecrement(&glNumberOfActiveComponents);// decrement global counter
}

// Implementation Of CSumSubtract's IUnknown's Methods
HRESULT CSumSubtract::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<ISum *>(this);
   else if (riid == IID_ISum)
      *ppv = static_cast<ISum *>(this);
   else if (riid == IID_ISubtract)
      *ppv = static_cast<ISubtract
   *>(this); else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

   reinterpret_cast<IUnknown *>(*ppv)->AddRef();

   return(S_OK);
}

ULONG CSumSubtract::AddRef(void)
{
   InterlockedIncrement(&m_cRef);
   return(m_cRef);
}

ULONG CSumSubtract::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);  // delete before posting WM_QUIT message

      if (glNumberOfActiveComponents == 0 && glNumberOfServerLocks == 0)
        PostMessage(ghwnd, WM_QUIT, (WPARAM)0, (LPARAM)0L);

      return(0);
   }
   return(m_cRef);
}
// Implementation Of ISum's Methods
HRESULT CSumSubtract::SumOfTwoIntegers(int num1, int num2, int *pSum)
{
   *pSum = num1 - num2; // done deliberately to identify the server is built for
   shantanu return(S_OK);
}

// Implementation Of ISubtract's Methods
HRESULT CSumSubtract::SubtractionOfTwoIntegers(int num1, int num2, int *pSubtract)
{
   *pSubtract = num1 + num2; // done deliberately to identify the server is built for
   shantanu return(S_OK);
}
```

```
// Implementation Of CSumSubtractClassFactory's Constructor Method
CSumSubtractClassFactory::CSumSubtractClassFactory(void)
{
   m_cRef = 1; // hardcoded initialization to anticipate possible failure of QueryInterface()
}

// Implementation Of CSumSubtractClassFactory's Destructor Method
CSumSubtractClassFactory::~CSumSubtractClassFactory(void)
{
   // no code
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's IUnknown's Methods
HRESULT CSumSubtractClassFactory::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<IClassFactory *>(this);
   else if (riid == IID_IClassFactory)
      *ppv = static_cast<IClassFactory
   *>(this); else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

   reinterpret_cast<IUnknown *>(*ppv)->AddRef();

   return(S_OK);
}

ULONG CSumSubtractClassFactory::AddRef(void)
{
   InterlockedIncrement(&m_cRef);

   return(m_cRef);
}

ULONG CSumSubtractClassFactory::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of CSumSubtractClassFactory's IClassFactory's Methods
HRESULT CSumSubtractClassFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid, void **ppv)
{
   // variable declarations
   CSumSubtract *pCSumSubtract =
   NULL; HRESULT hr;

   if (pUnkOuter != NULL)
      return(CLASS_E_NOAGGREGATION);

   // create the instance of component i.e. of CSumSubtract
   class pCSumSubtract = new CSumSubtract;
   if (pCSumSubtract == NULL)
      return(E_OUTOFMEMORY);

   // get the requested interface
   hr = pCSumSubtract->QueryInterface(riid, ppv);
```

```
    pCSumSubtract->Release();// anticipate possible failure of QueryInterface()

    return(hr);
}

HRESULT CSumSubtractClassFactory::LockServer(BOOL fLock)
{
    if (fLock)
        InterlockedIncrement(&glNumberOfServerLocks);
    else
        InterlockedDecrement(&glNumberOfServerLocks);

    if (glNumberOfActiveComponents == 0 && glNumberOfServerLocks == 0)
        PostMessage(ghwnd, WM_QUIT, (WPARAM)0, (LPARAM)0L);

    return(S_OK);
}

HRESULT StartMyClassFactories(void)
{
    // variable declaraions
    HRESULT hr;

    gpIClassFactory = new CSumSubtractClassFactory;
    if (gpIClassFactory == NULL)
        return(E_OUTOFMEMORY);

    gpIClassFactory->AddRef();

    // register the class factory in COM's private database
    hr = CoRegisterClassObject(CLSID_SumSubtract, static_cast<IUnknown *>(gpIClassFactory),
                CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE, &dwRegister);

    /* CoRegisterClassObject registers the CLSID for the server in what is called the class
    table (a different table than the running object table). When a server is registered in
    the class table, it allows the service control manager (SCM) to determine that it is not
    necessary to launch the class again, because the server is already running. Only if the
    server is not listed in the class table will the SCM check the registry for appropriate
    values and launch the server associated with the given CLSID. */

    if (FAILED(hr))
    {
        gpIClassFactory->Release();
        return(E_FAIL);
    }

    return(S_OK);
}
void StopMyClassFactories(void)
{
    // un-register the class
    factory if (dwRegister != 0)
        CoRevokeClassObject(dwRegister);

        /* CoRevokeClassObject revokes the class object (remove its registration) when all of
        the following are true:
    There are no existing instances of the object
    definition. There are no locks on the class object.
    The application providing services to the class object is not under user control (not
    visible to the user on the display). */

    if (gpIClassFactory != NULL)
        gpIClassFactory->Release();
}
```

```
DWORD GetParentProcessID(void)
{
   // variable declarations
   HANDLE hProcessSnapshot = NULL;
   BOOL bRetCode = FALSE;
   PROCESSENTRY32 ProcessEntry = { 0 };
   DWORD dwPPID;
   TCHAR szNameOfThisProcess[_MAX_PATH], szNameOfParentProcess[_MAX_PATH]; TCHAR
   szTemp[_MAX_PATH],/*szTemp2[_MAX_PATH],*/str[_MAX_PATH], *ptr = NULL;

   // first take current system snapshot
   hProcessSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
   0); if (hProcessSnapshot == INVALID_HANDLE_VALUE)
      return(-1);

   ProcessEntry.dwSize = sizeof(PROCESSENTRY32);

   // walk process hierarchy
   if (Process32First(hProcessSnapshot, &ProcessEntry))
   {
      GetModuleFileName(NULL, szTemp, _MAX_PATH);
      ptr = wcsrchr(szTemp, '\\');
      wcscpy_s(szNameOfThisProcess, ptr + 1);
      do
      {
         errno_t err;

         err = _wcslwr_s(szNameOfThisProcess, wcslen(szNameOfThisProcess) + 1);
         err = _wcsupr_s(ProcessEntry.szExeFile, wcslen(ProcessEntry.szExeFile) + 1);

         if (wcsstr(szNameOfThisProcess, ProcessEntry.szExeFile) != NULL)
         {
                              wsprintf(str, TEXT("Current Process Name = %s\nCurrent Process ID
                              = %ld\nParent Process ID = %ld\nParent Process Name = %s"),
                              szNameOfThisProcess, ProcessEntry.th32ProcessID,
                              ProcessEntry.th32ParentProcessID, ProcessEntry.szExeFile);

            MessageBox(NULL, str, TEXT("Parent Info"), MB_OK | MB_TOPMOST);

            dwPPID = ProcessEntry.th32ParentProcessID;
         }
      } while (Process32Next(hProcessSnapshot, &ProcessEntry));
   }

   CloseHandle(hProcessSnapshot);
   return(dwPPID);
}
```

4. ExeServer.reg

* Note that, we will register our out-of-process server under LocalServer32 and not under InprocServer32. Don't forget to register your server under HKEY_CLASSES_ROOT\WOW6432Node\CLSID path instread of HKEY_CLASSES_ROOT\CLSID path if your exe server is 32bit.

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{7ACCABF1-40CE-49a5-8703-0183CDE6B91F}]
@="ExeServer"
[HKEY_CLASSES_ROOT\CLSID\{7ACCABF1-40CE-49a5-8703-0183CDE6B91F}\LocalServer32]
@="C:\\ExeServer.exe"
```

5.2.1 CoRegisterClassObject function

Registers an EXE class object with OLE so other applications can connect to it.

```
HRESULT CoRegisterClassObject(
  _In_  REFCLSID  rclsid,
  _In_  LPUNKNOWN pUnk,
```

```
  _In_  DWORD     dwClsContext,
  _In_  DWORD     flags,
  _Out_ LPDWORD   lpdwRegister
);
```

Parameters

rclsid [in]: The CLSID to be registered.

pUnk [in]: A pointer to the **IUnknown** interface on the class object whose availability is being published.

dwClsContext [in]: The context in which the executable code is to be run. For information on these context values, see the **CLSCTX** enumeration.

flags [in]: Indicates how connections are made to the class object. For information on these flags, see the **REGCLS** enumeration.

```
typedef enum tagREGCLS {
  REGCLS_SINGLEUSE        = 0,
  /* After an application is connected to a class object with CoGetClassObject, the class object is removed from
   public view so that no other applications can connect to it. Thus, only 1 client can connect to the server. */

  REGCLS_MULTIPLEUSE      = 1,
  /* Multiple applications can connect to the class object through calls to CoGetClassObject. */
  REGCLS_MULTI_SEPARATE = 2,
  /* Useful for registering separate CLSCTX_LOCAL_SERVER and CLSCTX_INPROC_SERVER class factories
  through calls to CoGetClassObject. Used for registering separate out-of-process and in-process COM servers. */

  REGCLS_SUSPENDED        = 4,
  /* Suspends registration and activation requests for the specified CLSID until there is a call
  to CoResumeClassObjects. */

  REGCLS_SURROGATE        = 8
  /* The class object is a surrogate process used to run DLL servers. The class factory registered by the
  surrogate process is not the actual class factory implemented by the DLL server, but a generic class factory
  implemented by the surrogate. This generic class factory delegates instance creation and marshaling to the
  class factory of the DLL server running in the surrogate. */
} REGCLS;
```

lpdwRegister [out]: A pointer to a value (magic-cookie) that identifies the class object registered; later used by the CoRevokeClassObject function to revoke the registration.

Remarks

EXE object applications should call CoRegisterClassObject on startup. It can also be used to register internal objects for use by the same EXE or other code (such as DLLs) that the EXE uses. Only EXE object applications call CoRegisterClassObject. Object handlers or DLL object applications do not call this function — instead, they must implement and export the DllGetClassObject function.

At startup, a multiple-use EXE object application must create a class object (with the IClassFactory interface on it), and call CoRegisterClassObject to register the class object. Object applications that support several different classes (such as multiple types of embeddable objects) must allocate and register a different class object for each. Multiple registrations of the same class object are independent and do not produce an error. Each subsequent registration yields a unique key in lpdwRegister.

Multiple document interface (MDI) applications must register their class objects. Single document interface (SDI) applications must register their class objects only if they can be started by means of the /Embedding switch. The server for a class object should call CoRevokeClassObject to revoke the class object (remove its registration) when all of the following are true:

There are no existing instances of the object definition.
There are no locks on the class object.
The application providing services to the class object is not under user control (not visible to the user on the display).

After the class object is revoked, when its reference count reaches zero, the class object can be released, allowing the application to exit.

Note that CoRegisterClassObject calls IUnknown::AddRef and CoRevokeClassObject calls IUnknown::Release, so the two functions form an AddRef/Release pair.

5.2.2 CoRevokeClassObject function

Informs OLE that a class object, previously registered with the **CoRegisterClassObject** function, is no longer available for use.

```
HRESULT CoRevokeClassObject(_In_ DWORD dwRegister);
```

Parameters

dwRegister [in]: A token (magic-cookie) previously returned from the **CoRegisterClassObject** function.

Remarks

A successful call to CoRevokeClassObject means that the class object has been removed from the global class object table (although it does not release the class object). If other clients still have pointers to the class object and have caused the reference count to be incremented by calls to IUnknown::AddRef, the reference count will not be zero. When this occurs, applications may benefit if subsequent calls (with the obvious exceptions of AddRef and IUnknown::Release) to the class object fail.

Note that CoRegisterClassObject calls AddRef andCoRevokeClassObject calls Release, so the two functions form an AddRef/Release pair.

An object application must call CoRevokeClassObject to revoke registered class objects before exiting the program. Class object implementers should call CoRevokeClassObject as part of the release sequence. You must specifically revoke the class object even when you have specified the flags value REGCLS_SINGLEUSE in a call to CoRegisterClassObject, indicating that only one application can connect to the class object.

5.2. Create Proxy-Stub

Program structure:

1. ProxyStub.idl – IDL file to publish the interfaces in the world understandable format
2. ProxyStub.def – lists public and private exported methods
3. ProxyStub.rsp – Specifies a compiler response file. A response file can contain any commands that you would specify on the command line. This can be useful if your command-line arguments exceed 127 characters. It is not possible to specify the @ option from within a response file. That is, a response file cannot embed another response file.

ProxyStub.idl

```
import "unknwn.idl" ;
// ISum Interface
[
   object, uuid(8CC14612-CAB1-42dc-8E22-
   B88F5B2F0CC2), helpstring("ISum
   Interface"), pointer_default(unique)

]
```

```
interface IAddSubtract : IUnknown
{
   HRESULT SumOfTwoIntegers([in]int,[in]int,[in,out]int *);
};

// ISubtract Interface
[
   object, uuid(B16D9F59-BFA4-4c2d-90C6-
   7AC06B08BE91), helpstring("ISubtract
   Interface"), pointer_default(unique)

]

interface IMultiplyDevide : IUnknown
{
   HRESULT SubtractionOfTwoIntegers([in]int,[in]int,[in,out]int *);
};
```

ProxyStub.def

* Note that providing ordinal value i.e. @1, @2 for exported funcations in the def file are absolute and no longer needed.

```
LIBRARY ProxyStub.dll
DESCRIPTION 'MIDL Generated Proxy/Stub Dll File'
EXPORTS
      DllGetClassObject   @1 PRIVATE
      DllCanUnloadNow     @2 PRIVATE
      GetProxyDllInfo     @3 PRIVATE
      DllRegisterServer   @4 PRIVATE
      DllUnregisterServer @5 PRIVATE
```

ProxyStub.rsp

```
/h ProxyStubHeader.h
/iid ProxyStubGuids.c
/dlldata ProxyStubDlldata.c
/proxy ProxyStub.c
/out E:\PROGRA~1\MICROS~3\MYPROJ~1\COM_CO~1\06_EXE~1\WITHRE~1\EXESER~1\PROXYS~1
E:\PROGRA~1\MICROS~3\MYPROJ~1\COM_CO~1\06_EXE~1\WITHRE~1\EXESER~1\PROXYS~1\ProxyStub.idl
```

* Note: you can use "dir /x" DOS command to view DOS style directory names.

How to build proxy-server dll?

1. Create ProxyStub.idl, ProxyStub.def and ProxyStub.rsp files with above given content
2. Create a new directory - say "ProxyStub" - and copy above 3 files into it
3. From visual studio command/developer prompt, traverse to the ProxyStub directory and run following command to generate RPC code –

   > midl @ProxyStub.rsp // for 32-bit platform

   > midl /env x64 @ProxyStub.rsp // for 64-bit platform

* Note: Stub RPC code (.h and .c) created for 32-bit platform i.e. without using "/env x64" switch won't link for 64-bit build configurations (Debug & Release). You will get "unresolved external symbol ###_ProxyFileInfo" link time error. Similarly, stub RPC code (.h and .c) created for 64-bit platform using "/env x64" switch won't link for 32-bit build configurations. You will get "unresolved external symbol ###_ProxyFileInfo" link time error here too.

```
Developer Command Prompt for VS2015

c:\ProxyStub>midl /env x64 @ProxyStub.rsp
Microsoft (R) 32b/64b MIDL Compiler Version 8.00.0613
Copyright (c) Microsoft Corporation. All rights reserved.
64 bit Processing C:\PROXYS~1\ProxyStub.idl
ProxyStub.idl
64 bit Processing C:\Program Files (x86)\Windows Kits\10\include\10.0.10240.0\um\unknwn.idl
unknwn.idl
64 bit Processing C:\Program Files (x86)\Windows Kits\10\include\10.0.10240.0\shared\wtypes.idl
wtypes.idl
64 bit Processing C:\Program Files (x86)\Windows Kits\10\include\10.0.10240.0\shared\wtypesbase.idl
wtypesbase.idl
64 bit Processing C:\Program Files (x86)\Windows Kits\10\include\10.0.10240.0\shared\basetsd.h
basetsd.h
64 bit Processing C:\Program Files (x86)\Windows Kits\10\include\10.0.10240.0\shared\guiddef.h
guiddef.h

c:\ProxyStub>dir
 Volume in drive C is Windows
 Volume Serial Number is B25B-01F0

 Directory of c:\ProxyStub

01-07-2016  23:07    <DIR>          .
01-07-2016  23:07    <DIR>          ..
01-07-2016  23:11             9,541 ProxyStub.c
09-09-2010  11:55               272 ProxyStub.def
09-09-2010  11:55               525 ProxyStub.idl
01-07-2016  21:50               139 ProxyStub.rsp
01-07-2016  23:11               813 ProxyStubDlldata.c
01-07-2016  23:11             1,871 ProxyStubGuids.c
01-07-2016  23:11             6,571 ProxyStubHeader.h
               7 File(s)         19,732 bytes
               2 Dir(s)  93,454,114,816 bytes free
```

4. Notice three ".c" and one ".h" files created. Create a new Win32 DLL project using these 4 files in usual way.
5. Add ProxyStub.def file into the newly created project and add its reference under "Project Properties -> Configuration Properties -> Linker -> Input -> Module Definition File" option.
6. Now add reference (explicitly for each configuration and platform) of the **rpcrt4.lib** library file under the "Project Properties -> Configuration Properties -> Linker -> Input -> Additional Dependencies" option. Note that, if working with visual studio 2008 and older, you may also need to add reference of **rpcndr.lib** and **rpcns4.lib** library files along with rpcrt4.lib.
7. Add REGISTER_PROXY_DLL MACRO (explicitly for each configuration and platform) under "Project Properties -> Configuration Properties -> C/C++ -> Preprocessor -> Preprocessor Definitions" option. By defining the REGISTER_PROXY_DLL macro, while compiling Dlldata.c file, your proxy/stub marshaling DLL will automatically include default definitions for the DllMain, DllRegisterServer, and DllUnregisterServer functions. You can use these functions to self-register your proxy DLL in the system registry.
8. Now build the project. It should create proxy-stub RPC library, ProxyStub.dll in our case.

```
Output
Show output from: Build
1>------ Rebuild All started: Project: ProxyStub, Configuration: Release x64 ------
1>  ProxyStub.c
1>  ProxyStubDlldata.c
1>  ProxyStubGuids.c
1>     Creating library c:\users\rasu\documents\visual studio 2015\Projects\ExeServer\x64\Release\ProxyStub.lib and object c:\users\rasu
1>  Generating code
1>  All 8 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
1>  Finished generating code
1>  ProxyStubSumSubtract.vcxproj -> c:\users\rasu\documents\visual studio 2015\Projects\ExeServer\x64\Release\ProxyStub.dll
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

5.3. Client Implementation

Program structure:

1. ExeServerWithRegFile.h - to declare public interface consumable by clients
2. ClientOfExeServerWithRegFile.cpp – client side implementation to activate and consume the exe server

ExeServerWithRegFile.h

Refer to server's ExeServerWithRegFile.h file

ClientOfExeServerWithRegFile.cpp

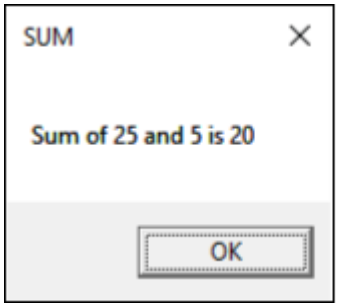
```
#define UNICODE

#include<windows.h>
#include<process.h>//for _wspawnlp() & exit()
#include"ExeServerWithRegFile.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
   // variable
   declarations WNDCLASSEX
   wndclass; HWND hwnd;
   MSG msg;
   TCHAR AppName[] = TEXT("Client");

   wndclass.cbSize = sizeof(wndclass);
   wndclass.style = CS_HREDRAW | CS_VREDRAW;
   wndclass.cbClsExtra = 0;
   wndclass.cbWndExtra = 0;
   wndclass.lpfnWndProc = WndProc;
   wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
   wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
   wndclass.hInstance = hInstance;
   wndclass.lpszClassName = AppName;
   wndclass.lpszMenuName = NULL;
   wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

   // register window class
   RegisterClassEx(&wndclass);

   // create window
   hwnd = CreateWindow(AppName,
      TEXT("Client Of Exe Server"),
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL);

   ShowWindow(hwnd, nCmdShow);
   UpdateWindow(hwnd);

   // message loop
   while (GetMessage(&msg, NULL, 0, 0))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   return(msg.wParam);
}
```

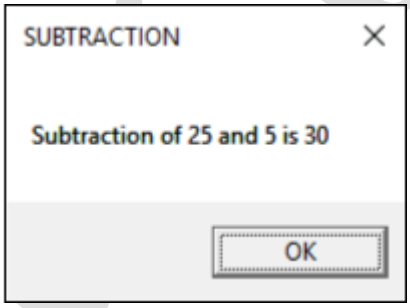
```
// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
   // variable declarations
   ISum *pISum = NULL;
   ISubtract *pISubtract =
   NULL; HRESULT hr;
   int error, n1, n2, n3; TCHAR
   szWinSysDir[255]; static TCHAR
   szPath[_MAX_PATH]; TCHAR
   str[255];

   switch (iMsg)
   {
   case WM_CREATE:
      // first register ProxyStub.dll
      GetSystemDirectory(szWinSysDir, 255);

      wsprintf(szPath, TEXT("%s\\regsvr32.exe"), szWinSysDir);

      error = _wspawnlp(P_WAIT, szPath, szPath, TEXT("/s"), TEXT("ProxyStub.dll"), NULL);

               if (error == -1)
      {
         MessageBox(hwnd, TEXT("Proxy/Stub Dll Can Not Be Registered"), TEXT("Error"), MB_OK);
         DestroyWindow(hwnd);
         exit(0);
      }

      // initialize COM library
      hr = CoInitialize(NULL);

      if (FAILED(hr))
      {
         MessageBox(hwnd, TEXT("COM library can not be initialized"), TEXT("COM Error"),
         MB_OK);

         DestroyWindow(hwnd);
         exit(0);
      }

      // get ISum Interface
      hr = CoCreateInstance(CLSID_SumSubtract, NULL, CLSCTX_LOCAL_SERVER, IID_ISum, (void
      **)&pISum);
      if (FAILED(hr))
      {
         MessageBox(hwnd, TEXT("Component Can Not Be Created"), TEXT("COM Error"), MB_OK);
         DestroyWindow(hwnd);

         exit(0);
      }

      n1 = 25;
      n2 = 5;
      pISum->SumOfTwoIntegers(n1, n2, &n3);
      wsprintf(str, TEXT("Sum of %d and %d is %d"), n1, n2, n3);

      MessageBox(hwnd, str, TEXT("SUM"), MB_OK);
      hr = pISum->QueryInterface(IID_ISubtract, (void **)&pISubtract);

      // get ISubtract Interface
      pISubtract->SubtractionOfTwoIntegers(n1, n2, &n3);
      wsprintf(str, TEXT("Subtraction of %d and %d is %d"), n1, n2, n3);

      MessageBox(hwnd, str, TEXT("SUBTRACTION"),
      MB_OK); DestroyWindow(hwnd);
      break;
   case WM_DESTROY:
```

```
        CoUninitialize();

        error = _wspawnlp(P_WAIT, szPath, szPath, TEXT("/u"), TEXT("/s"),
        TEXT("ProxyStub.dll"), NULL);

        if (error == -1)
              MessageBox(hwnd, TEXT("Proxy/Stub Dll Can Not Be Un-Registered"), TEXT("Error"),
              MB_OK);

        PostQuitMessage(0);
        break;
    }

    return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}
```

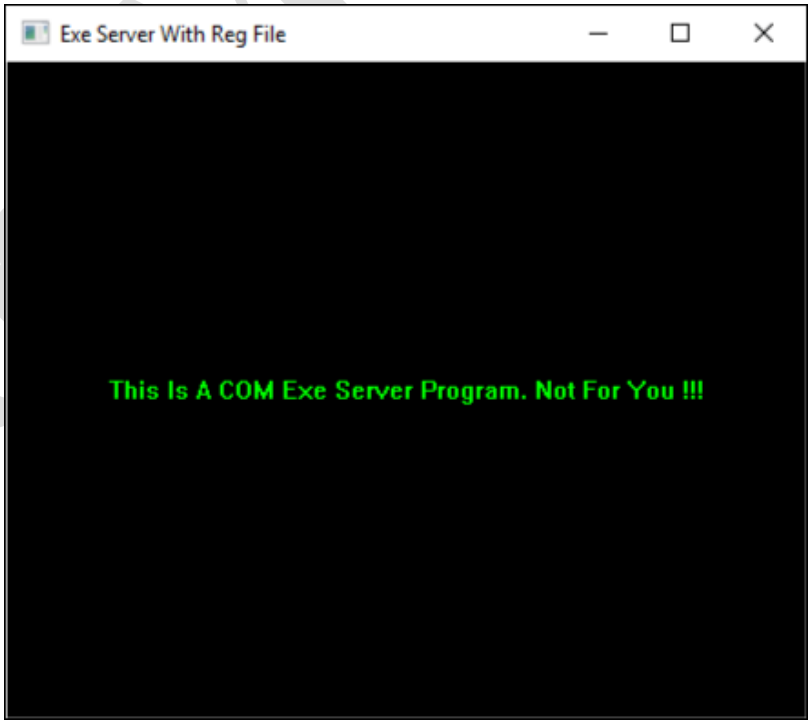
Client/Server execution output

(1) When users run the client program consuming exe server, following is what s/he sees.



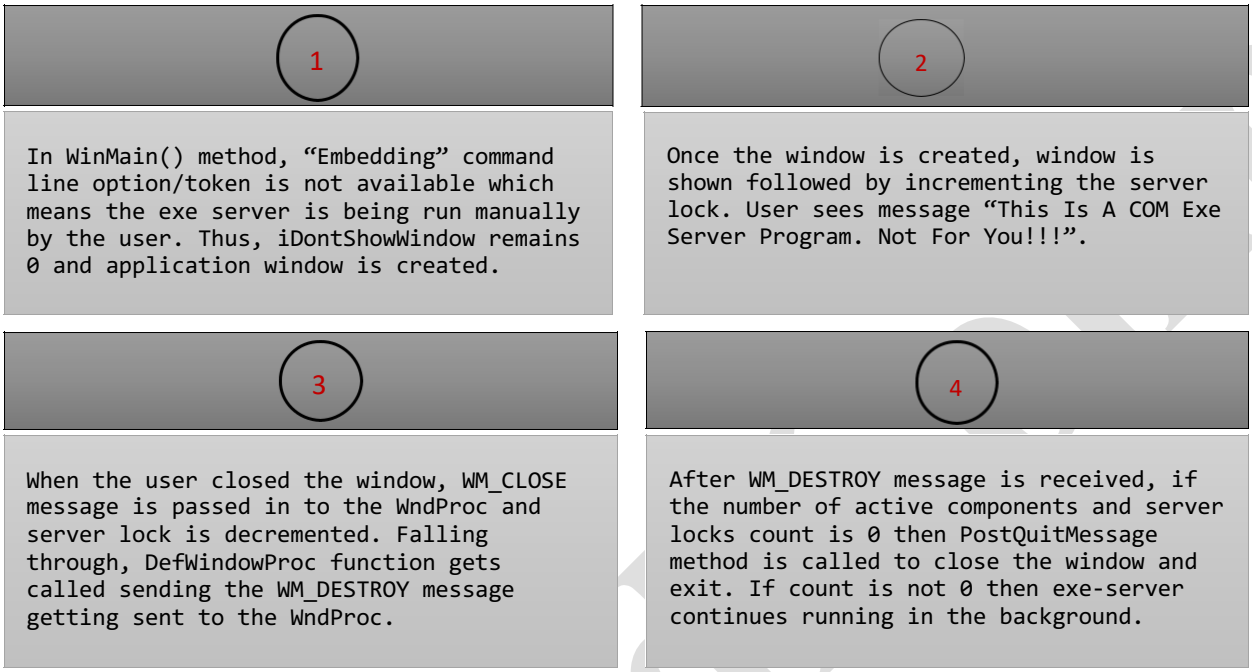


(2) When users run the exe server directly by double clicking on the ExeServer.exe file, following is what s/he sees.



5.4 Execution flow

Flow-1: When user manually executes the exe-server, following steps occur –



Flow-2: When a client application consumes exe-server without displaying to the user, following steps occur –

1	2
Client side CoGetClassObject API call searches for CLSID_SumSubtract. If found, value/path corresponding to the LocalServer32 key is read and exe is launched from the path.	COM sends "Embedding" command line argument to the exe-server. As the exe-server is being called by COM engine, StartMyClassFactories method is called instead of the UI code. Thus, user don't see any window/UI at all.
3	**4**
Inside StartMyClassFactories method, initialize the global classfactory instance followed by call to AddRef method. Now global classfactory instance is valid so call CoRegisterClassObject API to make it connectable to the client. CoRegisterClassObject registers CLSID in COM's private registry and in return a magic-cookie is given that points to a value in the COM's internal table. Maintain the magic-cookie which will be required later by CoRevokeClassObject API.	Classfactory pointer is instantiated and CreateInstance method is called using the pointer to initialize the component. In component's constructor, the global active component count is incremented as usual.
5	**6**
When it's time to close the connection with the server, client calls Release as usual. If active component and server count is 0 then WM_QUIT message is posted to the application.	Upon receiving the WM_DESTROY message, application gets out of the message loop and calls StopMyClassFactories method is called. Inside StopMyClassFactories method, CoRevokeClassObject API is called with the magic-cookie to inform OLE that a class object, previously registered with the CoRegisterClassObject function, is no longer available for use.

* Note that the proxy-stub dll marshals all the calls and data across client and server boundries using RPC/LPC protocols.

Page intentionally left blank

6. Service Control Manager (SCM)

The Service Control Manager (SCM) is the component of the COM Library responsible for locating class implementations and running them. The SCM ensures that when a client request is made, the appropriate server is connected and ready to receive the request. The SCM keeps a database of class information based on the system registry that the client caches locally through the COM library.

* Note: Windows NT also has a subsystem known as the Service Control Manager that is used to start logon-independent processes known as Services. So NT-SCM is different than COM-SCM.

When a client requests a COM object, the COM Library contacts the SCM on the local host. The SCM locates the appropriate COM server, which may be local or remote, and the server returns an interface pointer to the server's class factory. When the class factory is available, the COM Library or the client can use the class factory to create the requested object.

The Service Control Manager (SCM) handles the client request for an instance of a COM object. The following list shows the sequence of events:

A client requests an interface pointer to a COM object from the COM Library by calling a function such as CoCreateInstance with the CLSID of the COM object.

The COM Library queries the SCM to find the server that corresponds with the requested CLSID.

The SCM locates the server and requests the creation of the COM object from the class factory that is provided by the server.

If successful, the COM Library returns an interface pointer to the client.

The actions taken by the local SCM depend on the type of object server that is registered for the CLSID:

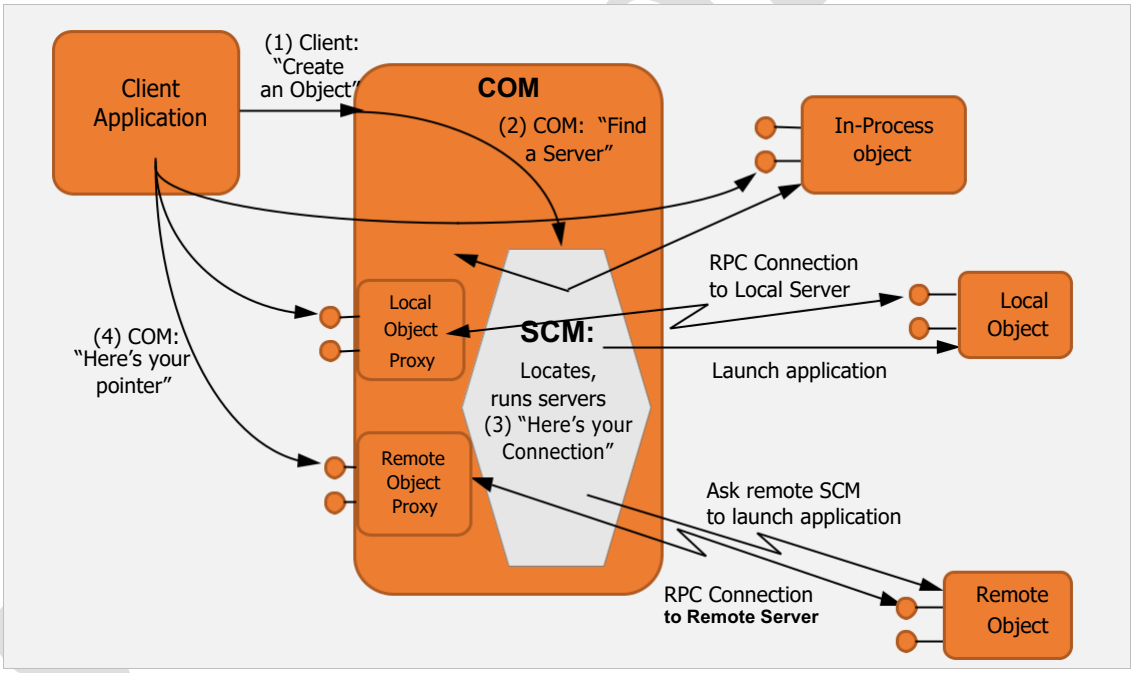


Figure 6-1: COM delegates responsibility of loading and launching servers to the SCM.

In-Process: The SCM returns the file path of the DLL containing the object server implementation. The COM library then loads the DLL and asks it for its class factory interface pointer.

Local: The SCM starts the local executable which registers a class factory on startup. That pointer is then available to COM.

Remote The local SCM contacts the SCM running on the appropriate remote machine and forwards the request to the remote SCM. The remote SCM launches the server which registers a class factory like the local server with COM on that remote machine. The remote SCM then maintains a connection to that class factory and returns an RPC connection to the local SCM which corresponds to that remote class factory. The local SCM then returns that connection to COM which creates a class factory proxy which will internally forward requests to the remote SCM via the RPC connection and thus on to the remote server.

Note that if the remote SCM determines that the remote server is actually an in-process server, it launches a "surrogate" server that then loads that in-process server. The surrogate does nothing more than pass all requests on through to the loaded DLL.

After the COM system connects a server object to a client, the client and object communicate directly. There is no added overhead from calling through an intermediary at run time.

COM has three activation models that can used to bring objects into memory to allow methods calls to be invoked. Clients can ask COM to

bind to the class object of a given class using CoGetClassObject API (absolutely necessary)
create new instances of a class based on a CLSID using CoCreateInstanceEx API
bring a persistent object to life based on the persistent state of the object using CoGetInstanceFromFile API.

Each of COM's three activation models use the services of the COM SCM. The SCM is the central rendezvous point for all activation requests on a particular machine.

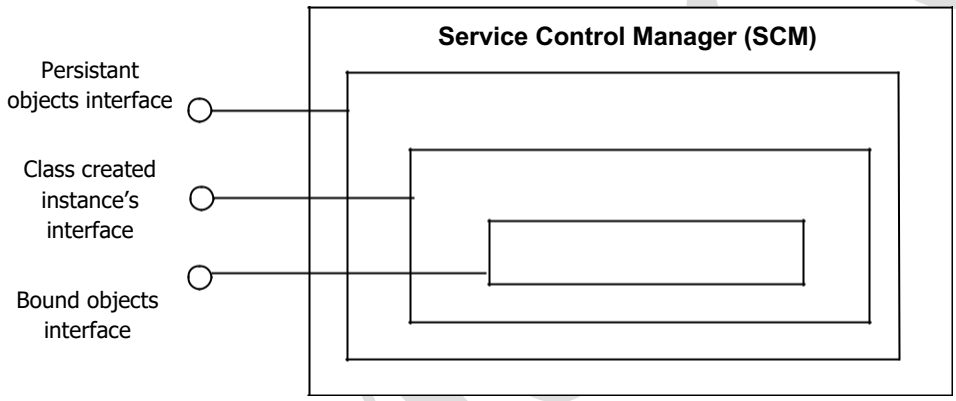


Figure 6-2: Service Control Manager

Page intentionally left blank

Page intentionally left blank

7. Security

As you know, COM uses RPC as its transport, COM security simply leverages the existing security infrastructure of RPC. COM security can be broken down into three categories:

Authentication: Authentication deals with ensuring that a message is authenticate, that is, that the sender is indeed who he says he is and that a given message is indeed from the sender.
Access control: Access control addresses who is allowed to access a server's objects and who is allowed to start a server process.
Token management: Token management is required to control which credentials are used when starting server processes and when executing inside a method call.

7.1 Authentication

COM provides somewhat reasonable defaults for each of these three aspects of security, if a programmer does nothing explicit with security. Most aspects of COM security can be configured by placing the correct information in the Registry. The DCOMCNFG.EXE program allows administrators to adjust most (but not all) of the settings related to COM security. For most (but not all) of these settings, the application programmer can elect to override the registry settings using explicit API functions. In general, most of the applications use a combination of DCOMCNFG.EXE settings and explicit API functions.

COM security uses the underlying RPC facilities for authentication and impersonation. RPC uses loadable transport modules to allow new network protocols to be added to the system after the fact. These transport modules are named using protocol sequences (e.g. ncacn_ip_tcp) that are mapped in the registry to a specific transport DLL. This allows third parties to install support for new transport protocols without modifying the COM library. Similarly, RPC supports loadable security packages to allow new security protocols to be added to the system after the fact. These security packages are named using integers that are mapped in the registry to a specific security package DLL. These DLLs must conform to the Security Support Provider Interface (SSPI), which was derived from the Internet Draft Standard GSSAPI (Generic Security Service Application Program Interface). Most of these DLLs provide their header files to programmers which mainly contain enumerated constants (enums) which mostly begin with "RPC_" tag.

Constant/value	Description
RPC_C_AUTHN_NONE = 0	No authentication.
RPC_C_AUTHN_DCE_PRIVATE = 1	Use Distributed Computing Environment (DCE) private key authentication.
RPC_C_AUTHN_DCE_PUBLIC = 2	DCE public key authentication (reserved for future use).
RPC_C_AUTHN_DEC_PUBLIC = 4	DEC public key authentication (reserved for future use).
RPC_C_AUTHN_GSS_NEGOTIATE = 9	Use the Microsoft Negotiate SSP. This SSP negotiates between the use of the NTLM and Kerberos protocol Security Support Providers (SSP).
RPC_C_AUTHN_WINNT = 10	Use the Microsoft NT LAN Manager (NTLM) SSP.
RPC_C_AUTHN_GSS_SCHANNEL = 14	Use the Schannel SSP. This SSP supports Secure Socket Layer (SSL), private communication technology (PCT), and transport level security (TLS).
RPC_C_AUTHN_GSS_KERBEROS = 16	Use the Microsoft Kerberos SSP.
RPC_C_AUTHN_DPA = 17	Use Distributed Password Authentication (DPA).
RPC_C_AUTHN_MSN = 18	Authentication protocol SSP used for the Microsoft Network (MSN).
RPC_C_AUTHN_DIGEST = 21	Windows XP or later: Use the Microsoft Digest SSP
RPC_C_AUTHN_NEGO_EXTENDER = 30	Windows 7 or later: Reserved. Do not use
RPC_C_AUTHN_MQ = 100	This SSP provides an SSPI-compatible wrapper for the Microsoft Message Queue (MSMQ) transport-level protocol.
RPC_C_AUTHN_DEFAULT = 0xffffffff	

CoInitializeSecurity API*:* Registers security and sets the default security values for the process.

```
HRESULT CoInitializeSecurity(
  _In_opt_ PSECURITY_DESCRIPTOR        pSecDesc, // access control
  _In_     LONG                        cAuthSvc, // # of sec pkgs (-1 == use default)
  _In_opt_ SOLE_AUTHENTICATION_SERVICE *asAuthSvc, // SSP array
  _In_opt_ void                        *pReserved1, // reserved, NULL
  _In_     DWORD                       dwAuthnLevel, // auto. AUTHN_LEVEL
  _In_     DWORD                       dwImpLevel,   // auto. IMP_LEVEL
  _In_opt_ void                        *pAuthList,  // reserved, used by COM
  _In_     DWORD                       dwCapabilities, // misc flags
  _In_opt_ void                        *pReserved3     // reserved NULL
);
```

COAUTHIDENTITY Structure: Contains a user name and password.

```
typedef struct _COAUTHIDENTITY {
  USHORT *User;              // The user's name.
  ULONG  UserLength;    // The length of the User string, without the terminating NULL.
  USHORT *Domain;    // The domain or workgroup name.
  ULONG  DomainLength; // The length of the Domain string, without the terminating NULL.
  USHORT *Password;     // The user's password in the domain or workgroup.
  ULONG  PasswordLength;  // The length of the Password string, without the terminating NULL.
  ULONG  Flags;              // Indicates whether the strings are Unicode strings.
} COAUTHIDENTITY;
```

Remark: COM does not persist the user's password information.

COAUTHINFO Structure: Contains the authentication settings used while making a remote activation request from the client computer to the server computer.

```
typedef struct _COAUTHINFO {
  DWORD          dwAuthnSvc;           // The authentication service to be used.
  DWORD          dwAuthzSvc;           // The authorization service to be used.
  LPWSTR         pwszServerPrincName;  // The server principal name to use with the auth
                                       // service.
  DWORD          dwAuthnLevel;         // The authentication level to be used.
  DWORD          dwImpersonationLevel; // The impersonation level to be used.
  COAUTHIDENTITY *pAuthIdentityData;   // A pointer to a COAUTHIDENTITY structure that
                                       // establishes
                                       // a nondefault client identity. If this parameter
                                       // is NULL,
                                       // the actual identity of the client is used.
  DWORD          dwCapabilities;       // Indicates additional capabilities of this proxy.
} COAUTHINFO;
```

Remarks: If pAuthInfo in COSERVERINFO is set to NULL, Snego will be used to negotiate an authentication service that will work between the client and server. However, a non-NULLCOAUTHINFO structure can be specified for pAuthInfo to meet any one of the following needs:

To specify a different client identity for computer remote activations. The specified identity will be used for the launch permission check on the server rather than the real client identity.

To specify that Kerberos, rather than NTLMSSP, is used for machine remote activation. A nondefault client identity may or may not be specified.

To request unsecure activation.

To specify a proprietary authentication service.

Specifying a COAUTHINFO structure allows DCOM activations to work correctly with security providers other than NTLMSSP. You can also specify additional security information used during remote activations for interoperability with alternate implementations of DCOM.
If you set dwAuthzSvc, pwszServerPrincName, dwImpersonationLevel, or dwCapabilities to incorrect values and call either CoGetClassObject or CoCreateInstanceEx, these functions do not return E_INVALIDARG or a similar error. Default values are used instead of the incorrect values.

COSERVERINFO Structure: Identifies a remote computer resource to the activation functions.

```
typedef struct _COSERVERINFO {
  DWORD      dwReserved1;   // This member is reserved and must be 0.
  LPWSTR     pwszName;      // The name of the computer.
  COAUTHINFO *pAuthInfo;    // A pointer to a COAUTHINFO structure to override the default
                            // activation security for machine remote activations.
  DWORD      dwReserved2;   // This member is reserved and must be 0.
} COSERVERINFO;
```

Remarks: The COSERVERINFO structure is used primarily to identify a remote system in object creation functions. Computer resources are named using the naming scheme of the network transport. By default, all UNC ("*server*" or "*server*") and DNS names ("*domain*.com", "*example*.microsoft.com", or "135.5.33.19") names are allowed.
If pAuthInfo is set to NULL, Snego will be used to negotiate an authentication service that will work between the client and server. However, a non-NULL COAUTHINFO structure can be specified for pAuthInfo to meet any one of the following needs:

To specify a different client identity for computer remote activations. The specified identity will be used for the launch permission check on the server rather than the real client identity.

To specify that Kerberos, rather than NTLMSSP, is used for machine remote activation. A nondefault client identity may or may not be specified.

To request unsecure activation.

To specify a proprietary authentication service.

If pAuthInfo is not NULL, those values will be used to specify the authentication settings for the remote call. These settings will be passed to the RpcBindingSetAuthInfoEx function.
If the *pAuthInfo* parameter is NULL, then *dwAuthnLevel* can be overridden by the authentication level set by the CoInitializeSecurity function. If the CoInitializeSecurity function isn't called, then the authentication level specified under the AppID registry key is used, if it exists.

SOLE_AUTHENTICATION_SERVICE Structure: Identifies an authentication service that a server is willing to use to communicate to a client.

```
typedef struct tagSOLE_AUTHENTICATION_SERVICE {
  DWORD   dwAuthnSvc;       // The authentication service.
  DWORD   dwAuthzSvc;       // The authorization service.
  OLECHAR *pPrincipalName;  // The principal name to be used with the authentication service.
  HRESULT hr;               // When used in CoInitializeSecurity, set on return to indicate
                            // the status of the call to register the authentication services.
} SOLE_AUTHENTICATION_SERVICE, *PSOLE_AUTHENTICATION_SERVICE;
```

EOLE_AUTHENTICATION_CAPABILITIES Structure: Specifies various capabilities in CoInitializeSecurity and IClientSecurity::SetBlanket (or its helper function CoSetProxyBlanket).

```
typedef enum tagEOLE_AUTHENTICATION_CAPABILITIES {
  EOAC_NONE                = 0,
  EOAC_MUTUAL_AUTH         = 0x1,
```

```
  EOAC_STATIC_CLOAKING    = 0x20,
  EOAC_DYNAMIC_CLOAKING   = 0x40,
  EOAC_ANY_AUTHORITY      = 0x80,
  EOAC_MAKE_FULLSIC       = 0x100,
  EOAC_DEFAULT            = 0x800,
  EOAC_SECURE_REFS        = 0x2,
  EOAC_ACCESS_CONTROL     = 0x4,
  EOAC_APPID              = 0x8,
  EOAC_DYNAMIC            = 0x10,
  EOAC_REQUIRE_FULLSIC    = 0x200,
  EOAC_AUTO_IMPERSONATE   = 0x400,
  EOAC_NO_CUSTOM_MARSHAL  = 0x2000,
  EOAC_DISABLE_AAA        = 0x1000
} EOLE_AUTHENTICATION_CAPABILITIES;
```

IServerSecurity Interface: IServerSecurity may be used to impersonate the client during a call, even on other threads within the server. IServerSeciruty:QueryBlanket and IServerSecurity::ImpersonateClient may only be called before the call completes. IServerSecurity::RevertToSelf may be called at any time. The interface pointer must be released when it is no longer needed. Unless the server wishes to impersonate the client on another thread, there is not reason to keep an IServerSecurity past the end of the call, since it will at that point no longer support IServerSecurity::QueryBlanket.

```
interface IServerSecurity : IUnknown {
    HRESULT QueryBlanket(RPC_AUTHZ_HANDLE* Privs, WCHAR** ServerPrincName, DWORD*
        AuthnLevel, DWORD* AuthnSvc, DWORD* AuthzSvc );
    HRESULT ImpersonateClient(void);
    HRESULT RevertToSelf(void);
    };
```

IServerSecurity::QueryBlanket

```
HRESULT IServerSecurity::QueryBlanket(Privs, ServerPrincName, AuthnLevel, AuthnSvc, AuthzSvc);
```

This method is used by the server to find out about the client that invoked one of its methods. CoGetCallContext with IID_ISeverSecurity returns an IServerSecurity interface for the current call on the current thread. This interface pointer may be used on any thread and calls to it may succeed until the call completes.

Argument	Type	Description
Privs	RPC_AUTHZ_HANDLE*	Returns a pointer to a handle to the privilege information for the client application. The format of the structure is authentication service specific. The application should not write or free the memory. The information is only valid for the duration of the current call. NULL may be passed if the application is not interested in this parameter.
ServerPrincName	WCHAR*	This parameter indicates the principal name the client specified. It is a copy allocated with CoTaskMemAlloc. The application must call CoTaskMemFree to release it. NULL may be passed if the application is not interested in this parameter.
AuthnLevel	DWORD*	This parameter indicates the authentication level. NULL may be passed if the application is not interested in this parameter.
AuthnSvc	DWORD*	This parameter indicate the authentication service the client specified. NULL may be passed if the application is not interested in this parameter.
AuthzSvc	DWORD*	This parameter indicates the authorization service. NULL may be passed if the application is not interested in this parameter.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.
	E_OUTOFMEMORY	Insufficient memory to create one or more out-parameters.

IServerSecurity::ImpersonateClient

```
HRESULT IServerSecurity::ImpersonateClient();
```

This method allows a server to impersonate a client for the duration of a call. The server may impersonate the client on any secure call at identify, impersonate, or delegate level. At identify level, the server may only find out the clients name and perform ACL checks; it may not access system objects as the client. At delegate level the server may make off machine calls while impersonating the client. The impersonation information only lasts till the end of the current method call. At that time IServerSecurity::RevertToSelf will automatically be called if necessary.

Impersonation information is not normally nested. The last call to any Win32 impersonation mechanism overrides any previous impersonation. However, in the apartment model, impersonation is maintained during nested calls. Thus if the server *A* receives a call from *B*, impersonates, calls *C*, receives a call from *D*, impersonates, reverts, and receives the reply from *C*, the impersonation will be set back to *B*, not *A*.

If IServerSecurity::ImpersonateClient is called on a thread other then the one that received the call, the impersonation will not automatically be revoked. It will be valid past the end of the call. However, IServerSecurity::ImpersonateClient must be called before the original call completes.

Argument	Type	Description
Returns	S_OK	Success.
	E_FAIL	The caller can not impersonate the client identified by this ISeverSecurity object.

IServerSecurity::RevertToSelf

```
HRESULT IServerSecurity::RevertToSelf();
```

This method restores the authentication information on a thread to the process's identity.

In the apartment model, IServerSecurity::RevertToSelf only affects the current method invocation. If there are nested method invocations, they each may have there own impersonation and COM will correctly restore the impersonation before returning to them (regardless of whether or not IServerSecurity::RevertToSelf was called).

IServerSecurity::RevertToSelf may be called on threads other then the one that received the call. IServerSecurity::RevertToSelf may be called after the call completes. Calls to IServerSecurity::RevertToSelf that are not matched with an IServerSecurity::ImpersonateClient call will fail.

Argument	Type	Description
Returns	S_OK	Success.
	E_FAIL	This call was not preceded by a call to IServerSecurity::ImpersonateClient on this thread of execution.

IClientSecurity Interface: IClientSecurity gives the client control over the call-security of individual interfaces on a remote object. All proxies generated by the COM MIDL compiler support the IClientSecurity interface. If QueryInterface for IClientSecurity fails, either the object is implemented in-process or it is remoted by a custom marshaler which does not support security (a custom marshaler may support security by offering the IClientSecurity interface to the client). The proxies passed as parameters to an IClientSecurity method must be from the same object as the IClientSecurity interface.

```
interface IClientSecurity : IUnknown {
        HRESULT QueryBlanket(void* pProxy, DWORD* pcbAuthnSvc, SOLE_AUTHENTICATION_SERVICE*
        pasAuthnSvc, RPC_AUTH_IDENTITY_HANDLE** ppAuthInfo, DWORD* AuthnLevel); HRESULT
        SetBlanket(void* pProxy, DWORD AuthnSvc, WCHAR* ServerPrincName,
        RPC_AUTH_IDENTITY_HANDLE* pAuthInfo, DWORD AuthnLevel, DWORD AuthzSvc);
   HRESULT CopyProxy(void* pProxy, REFIID riid, void** ppCopy);
};
```

IClientSecurity::QueryBlanket

```
HRESULT IClientSecurity::QueryBlanket(pProxy, pcbAuthnSvc, pasAuthnSvc, ppAuthInfo,
AuthnLevel);
```

This method returns authentication information. This method is called by the client to find out what authentication information COM will use on calls made from the specified proxy.

Argument	Type	Description
pProxy	void*	This parameter indicates the proxy to query.
pcbAuthnSvc	DWORD*	This parameter indicates the number of entries in the array pasAuthSvc.
pasAuthnSvc	SOLE_AUTHENTICATION_SERVICE*	This parameter is an array of authentication service, principal name pairs. The first entry is the one that COM will use to make calls to the server. The array is allocated with CoTaskMemAlloc and the application must free it by calling CoTaskMemFree.
ppAuthInfo	RPC_AUTH_IDENTITY_HANDLE**	This parameter returns the value passed to CoSetProxyAuthenticationInfo. It may be NULL if you do not care.
AuthnLevel	DWORD*	This parameter returns the current authentication level. It may be NULL if you do not care.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.
	E_OUTOFMEMORY	Insufficient memory to create the pasAuthnSvc out-parameter.

IClientSecurity::SetBlanket

```
HRESULT IClientSecurity::SetBlanket(pProxy, AuthnSvc, ServerPrincName, pAuthInfo, AuthnLevel,
AuthzSvc);
```

This method sets the authentication information that will be used to make calls on the specified proxy. The values specified here override the values chosen by automatic security. Calling this method changes the security values for all other users of the specified proxy. Use IClientSecurity::CopyProxy to make a private copy.
By default the authentication service and principal name is set to a list of authentication service and principal name pairs that were registered on the server. When this method is called COM will forget the default list. By default COM will try one principal name from the list of authentication services available on both machines. It will not retry if that principal name fails.

If pAuthInfo is not set, it defaults to the logged in id. AuthnLevel and AuthzSvc default to the values specified to CoInitializeSecurity. If CoInitializeSecurity is not called, they default to RPC_C_AUTHN_LEVEL_NONE and RPC_C_AUTHZ_NONE.

Security information will often be ignored if set on local interfaces. For example, it is legal to set security on the IClientSecurity interface. However, since that interface is supported locally, there is no need for security. IUnknown and IMultiQuery are special cases. The local implementation makes remote calls to support these interfaces. The local implementation will use the security settings for those interfaces.

Argument	Type	Description
pProxy	void*	This parameter indicates the proxy to set.
AuthnSvc	DWORD	This parameter indicates the authentication service. It may be RPC_C_AUTHN_NONE if no authentication is required. It may be RPC_C_AUTHN_DONT_CHANGE if you do not want to change the current value.
ServerPrincName	WCHAR*	This parameter indicates the server principal name. It may be NULL if you don't want to change the current value.
pAuthInfo	RPC_AUTH_IDENTITY_HANDLE*	This parameter sets the identity of the client. It is authentication service specific. Some authentication services allow the application to pass in a different user name and password. COM keeps a pointer to the memory passed in until

		COM is uninitialized or a new value is set. If NULL is specified COM uses the current identity (whether the logged in or impersonated id).
AuthnLevel	DWORD	This parameter specifies the authentication level. It may be RPC_C_AUTHN_LEVEL_DONT_CHANGE if you do not want to change the current value.
AuthzSvc	DWORD	This parameter specifies the authorization level. It may be RPC_C_AUTHZ_DONT_CHANGE is you do not want to change the current value.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments is invalid.

IClientSecurity::CopyProxy

```
HRESULT IClientSecurity::CopyProxy(pProxy, riid, ppCopy)
```

This method makes a copy of the specified proxy. Its authentication information may be changed without affecting any users of the original proxy. The copy has the default values for the authentication information. The copy has one reference and must be released.

Local interfaces may not be copied. IUnknown, IMultiQuery, and IClientSecurity are examples of existing local interfaces.

Argument	Type	Description
pProxy	void*	This parameter indicates the proxy to copy.
riid	REFIIID	Identifies the proxy to return.
ppCopy	void**	The copy is returned to this parameter.
Returns	S_OK	Success.
	E_NOINTERFACE	The interface riid is not supported by this object.

7.2 Access Control

Each COM process can protect itself against unauthorized access. COM addresses access control at two levels: *launch permissions* and *access permissions*. Launch permissions are used to determine which users can start server processes when making activation calls to the SCM. Access permissions determine which users can access a process's objects once the server is started. Both types of access control can be configured using DCOMCNFG.EXE, but only access permissions can be specified programmatically at runtime. Instead, launch permission is enforced by the SCM at activation time.

When the CM determines that a new server process must be started, it attempts to get an NT SECURITY_DESCRIPTOR that describes which users are allowed to start the server process. The SCM first checks the class's AppID for an explicit launch permission setting. This setting comes in the form of a serialized self-relative NT security descriptor that is stored at the AppID's LaunchPermission names value.

TRUSTEE structure

The TRUSTEE structure identifies the user account, group account, or *logon session* to which an *access control entry* (ACE) applies. The structure can use a name or a *security identifier* (SID) to identify the trustee. Access control functions, such as SetEntriesInAcl and GetExplicitEntriesFromAcl, use this structure to identify the logon account associated with the access control or audit control information in an EXPLICIT_ACCESS structure.

```
typedef struct _TRUSTEE {
  PTRUSTEE                  pMultipleTrustee;
  MULTIPLE_TRUSTEE_OPERATION MultipleTrusteeOperation;
  TRUSTEE_FORM              TrusteeForm;
  TRUSTEE_TYPE              TrusteeType;
  LPTSTR                    ptstrName;
} TRUSTEE, *PTRUSTEE;
```

ACTRL_ACCESS_ENTRY structure

Contains access-control information for a specified trustee. This structure stores information equivalent to the access-control information stored in an ACE.

```
typedef struct _ACTRL_ACCESS_ENTRY {
  TRUSTEE       Trustee;
  ULONG         fAccessFlags;
  ACCESS_RIGHTS Access;
  ACCESS_RIGHTS ProvSpecificAccess;
  INHERIT_FLAGS Inheritance;
  LPCTSTR       lpInheritProperty;
} ACTRL_ACCESS_ENTRY, *PACTRL_ACCESS_ENTRY;
```

ACTRL_PROPERTY_ENTRY structure

Contains a list of access-control entries for an object or a specified property on an object.

```
typedef struct _ACTRL_PROPERTY_ENTRY {
  LPCTSTR                  lpProperty;
  PACTRL_ACCESS_ENTRY_LIST pAccessEntryList;
  ULONG                    fListFlags;
} ACTRL_PROPERTY_ENTRY, *PACTRL_PROPERTY_ENTRY;
```

ACTRL_ACCESS structure

Contains an array of access-control lists for an object and its properties.

```
typedef struct _ACTRL_ACCESS {
  ULONG                 cEntries;
  PACTRL_PROPERTY_ENTRY pPropertyAccessList;
} ACTRL_ACCESS, *PACTRL_ACCESS;
```

IAccessControl interface

Enables the management of access to objects and properties on the objects.

When to implement

Distributed COM provides an implementation of the IAccessControl interface. COM servers can use this implementation to help protect their objects from unauthorized access. To get a pointer to this implementation, call CoCreateInstance, specifying CLSID_DCOMAccessControl as the CLSID. This implementation supports the IPersist interface to save the state of the access control object. The implementation of IAccessControl provided by COM calls built-in access control functions such as OpenThreadToken and AccessCheck. If you decide to implement IAccessControl yourself, you can also call these access control functions. However, because IAccessControl methods take access information in a different format than the built-in access control functions do, your implementation must be able to convert from one format to the other as necessary. If you decide to implement IAccessControl and pass your implementation to CoInitializeSecurity, be sure that it is completely thread-safe, because COM can call it on any thread, at any time. In addition to the COM implementation of IAccessControl, another implementation is supplied for storage and Directory Service objects.

When to use

Call methods of the IAccessControl interface to manage access to objects and properties on the objects and to obtain access information. This interface is primarily used to set process wide security with a call to CoInitializeSecurity, specifying EOAC_ACCESS_CONTROL as the capability flag, and providing a pointer to an instance of IAccessControl as the first (*pVoid*) parameter. COM then calls IAccessControl methods to determine access rights. IAccessControl should be used only to manage access rights. To manage launch permissions, use DCOMCNFG or set the LaunchPermission value under the AppID registry key. The IAccessControl interface inherits from the IUnknown interface.

7.3 Token Management

Under Windows NT, each process has an access token that represents the credentials of a security principal. This access token is created at process initialization time and contains various pieces of information about a user, including

the user's NT security identifier (SID), the list of groups the user belongs to, as well as the list of privileges the user holds (e.g. whether the user can shut down the system, whether the user can change the system clock). When a process attempts to gain access to secure kernel resources (e.g. files, registry keys, semaphores), the NT Security Reference Monitor uses the caller's token for auditing and access control purposes.

When an ORPC request message arrives at a process, COM arranges for the corresponding method call to execute on either an RPC thread (in the case of MTA-based objects) or a user-created thread (in the case of STA-based objects). In either case, the method executed using the access token of the process. It allows object implementers to predict what privileges and rights their objects will have irrespective of which user has issued the request. However, it is occasionally useful for a method to execute using the credentials of the client that invoked the method, either to restrict or enhance the normal rights and privileges of the object. To support this style of programming, Windows NT allows access token to be assigned to individual threads. When a thread has its own token, the Security Reference Monitor does not use the process token. Instead, the token that is assigned to the thread is used to perform auditing and access control. Although, it is possible to create tokens programmatically and assign them to threads, COM provides a much more direct mechanism for creating a token based on the ORPC request being serviced by the current thread. This mechanism is exposed to object implementers via the call context object.

SCM configures and controls token management. When RPC arrived Microsoft created its own ORPC protocol (Object Remote Procedure Call). This protocol is implemented in "call context" object which can be obtained using CoGetComContext COM API.

Note that, interface mention in Authentication i.e. IServerSecurity implements this call context object. You can see all these things in active running stage in a utility called NTLM (NT Lan Manager). When SCM starts server process it assigns a token to the process based on configuration of Server Processes AppId and 'RunAs' value. If server is not configured then obviously it doesn't have any 'RunAs' value and then it doesn't have any distributed access.

Though server doesn't have 'RunAs' value, the remote client can run this process but without COM facilities. To keep such process away from breeching the security, SCM starts these processes (not having RunAs value) according to clients activation requests (in COSERVERINFO). Such type of execution is called "As Activator" type activation.

Page intentionally left blank

8. Data Types & Component Categories

8.1 BSTR (Binary String)

The BSTR (*Binary String* or *Basic String*) string type must be used in all interfaces that will be used from Visual Basic or Java. BSTRs are length-prefixed, null-terminated strings of OLECHARs. The length prefix indicates the number of bytes the string consumes (not including the terminating null) and is stored as a four-byte integer that immediately precedes the first character of the string.

It is a specialized data type used by COM universally to maintain the language neutrality. Internally, it is a 32-bit, UNICODE compliant pointer to a wide-character string i.e. wchar_t.

```
BSTR szBstr[] = L"COM is better C++"; // incorrect
```

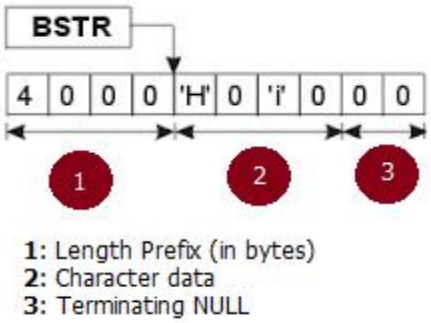
Above declaration is incorrect because compiler cannot guess the length of string and thus cannot put it at the beginning of the array.

```
wchar_t szString[] = L"COM is Better C++";
BSTR szBstr;
szBstr = SysAllocString(szString);
SysFreeString(szBstr); // deallocate (free) the memory allocated originally
           using // SysAllocString
```

Points to remember:

- Methods SysAllocString() and SysFreeString() are COM compliant.
- Use wsprintf to convert BSTR type string back to wchar_t type

Memory allocation in BSTR:
Following image shows the string "Hi" as a BSTR.



There are 2 COM compliant functions for memory allocation and de-allocation viz. CoTaskMemAlloc CoTaskMemFree respectively.

If you want to convert BSTR type string to WCHAR_T type string then use wsprintf function. Conversion is possible as BSTR is nothing but internally a pointer to a wide character string. If you want to convert WCHAR_T to ANSII formatted string, use wide-char to mbcs or mbcs to wide-char format as needed e.g. wcstombs.

8.2 VARIANT

COM predefines one common discriminated union for use with Visual Basic. This union in called VARIANT and can hold instances or references to a subset of the base type supported by IDL.

Variant is a typedef for a structure which is internally made-up of different structures and unions.

- As name suggest, this data type is used to store different types of variables in one common way. VB internally uses this data type massively.
- This is the only data type by which Automation interfaces and dual interfaces can communicate with each other.
- There is yet another variation of VARIANT data type called as VARIANTARG. Both are same.
- VARIANT data type frees us from static type checking because type checking for variant data type is completely dynamic i.e. Runtime.

This idea originally came from smalltalk language.

Each supported type has a corresponding discriminator value:

VT_EMPTY : nothing
VT_NULL : SQL style NULL
VT_I2 : short
VT_I4 : long
VT_R4 : real/float
VT_R8 : real/double
VT_I1 : character
VT_UI2 : unsigned short
VT_UI4 : unsigned long
VT_CY : CY/currency (64 bit value)
VT_DATE : date (double type)
VT_BSTR : BSTR/binary string
VT_INT : integer
VT_ERROR : HRESULT
VT_BOOL : VARIANT_BOOL, one for TRUE and zero for FALSE
VT_UNKNOWN : IUNKNOWN *
VT_DISPATCH : IDISPATCH *
VT_VARIANT : VARIANT *
VT_DECIMAL : 16 byte fixed point
VT_UI1 : opaque byte

The following two flags can be combined with these tags to indicate that the variant contains either a reference or an array of the specified type.

VT_ARRAY : indicates variant contains a SAFEARRAY
VT_BYREF : indicates variant is a reference

COM provides several API functions for managing VARIANTs:

```
void VariantInit(VARIANTARG *) ;
HRESULT VaraiantClear(VARIANTARG *);

VARIANT var;
VariantInit(&Var);      // initialize VARIANT
Var.VT = VT_T4;      // set discriminator
Var.LVAL = 100;      // set union
   .
   .
   .
   .
/* USE VARIANT IN AUTOMATION METHOD */
   .
   .
   .
   .
VariantClear(&Var);     // free any resources in VARIANT
```

COM also provides MACROs to initialize variant members.

```
V_VT(&var) = VT_I4; // MACRO – V_VT
V_I4(&var) = 100; // MACRO - V_I4
```

8.3 Component Categories

Each COM component is given a unique identifier (a GUID) that the COM runtime environment, called the Service Control Manager (SCM), uses to identify the component during execution. COM systems also often use another area of the registry, called component categories. Conceptually, a component category is a convenient way to logically group together classes that provide a certain type of functionality. Generally, all the classes in a particular category will support an agreed set of interfaces, although sometimes the classes simply conform to a semantic description of functionality. Any application can read the contents of a component category at run time to gather information about which classes support certain functionality without that application needing to know precise class names in advance.

Component categories enable a software component's abilities and requirements to be identified by entries in the registry.

All the component categories on a machine can be found by opening the appropriate registry key.

```
HKEY_CLASSES_ROOT\Component Categories
```

Each subkey identifies a category; each subkey name is a unique identifier or GUID, which is referred to as a CATID. Each CATID key contains a descriptive name for the category as a string data value.

For example, when using Visual Basic, if you click on Project\components... menu item, the *component* property sheet gets displayed listing all ActiveX components that VB supports. Here, we may think that QueryInterface is being called to find out whether particular component's interface provides needed facility or not. In that case, VB has to go through all the registry entries and call all components' QueryInterface to find out the supporting interfaces. But this will take enormous time and thus not practical. The way it works is using to the *component categories* where each component of certain type like control, document, automation, script etc. has registered itself in the system registry. If component belongs to a specific category then then client can identify their required functionality from the registry. So it is important that the component developer registers its component by defining registry key under predefined or a new category/group in the system registry.

A component category is a group of logically related COM classes that share a common category ID or CATID. CATIDs are GUIDs that are stored in the Registry as attributes of a class. Each class can have two subkeys: Implemented Categories and Required Categories. A CATID is a unique identifier for a particular component category.

For example, there is a predefined CATID {40FC6ED4-2438-11CF-A3D8-080036F12502} for the component category *control*. You can add your component of type control under this CATID or you can also define your own CATID too. But it is strongly recommended that you register your component under pre-defined CATID rather than defining new ones. Every CATID has an associated human readable *help string* which client can read and select desired component. This is the same *help string* which VB displays in the component property sheet.

CATID allows client to search for required functionality supporting component. Additionally, a component can also declare that it requires some particular functionality of client. Thus component can also choose acceptable clients using required CATID key, which are defined in registry under particular CATID as *Required Categories*, where container's (i.e. client) requirement are defined.

When you open the system registry using regedit.exe system tool and search for this CATID, then after opening it you will find two subsections under it.

1. Implemented Categories: components are listed under this subsection implements the functionality of this certain category.
2. Required categories: the requirements or constraints on the clients to use these implemented categories and such valid clients are listed under this subsection, so CATID works both ways.

If CATID is assigned as default GUID to a component, then the clients can instantiate the component without knowing its CLSID. This implies that CATID can be passed as the first parameter to the CoCreateInstantiate API function to get the IUnknown pointer followed by call to QueryInterface to get the required Interface

For instance, suppose a spell checker has a defined CATID say CATID_SpellChecker, then following call is sufficient to create its object

```
HRESULT hr = CoCreateInstance(CATID_SpellChecker, NULL,
CLSCTX_INPROC_SERVER, IID_IUnknown, (void **)&pIUnknown);
```

And now to get required spellchecker interface, following call is sufficient:

```
Hr = pIUnknown->QueryInterface(IID_ISpellChecker, (void **)&pISpellChecker);
```

Here first registry is checked for the `CATID_SpellChecker` CATID and then it was mapped to the actual CLSID of the Spell Checker component. When object is created, `ISpellChecker` interface is returned.

8.4 Component Categories Manager

To facilitate the handling of component categories and to guarantee consistency of the registry, the system provides the Component Categories Manager, a COM object with a CLSID of CLSID_StdComponentCategoriesMgr. It is implemented in C:\Windows\System32\comcat.dll library. To use this library programmatically, include comcat.h header file in the code. This COM object provides the following interfaces:

1. ICatInformation: provides methods for obtaining information about the categories implemented or required by a certain class and provides information about the categories registered on a given machine.

   When to implement: You do not need to implement this interface. The component category manager, a system-provided COM object that can be instantiated by using CoCreateInstance, implements ICatInformation.

   When to use: Call the methods of ICatInformation to obtain a listing of available categories, enumerate classes that belong to a particular category, and determine if a class belongs to a specific category.

   The ICatInformation interface has these methods.

Method	Description
EnumCategories	Retrieves an enumerator for the component categories registered on the system.
EnumClassesOfCategories	Retrieves an enumerator for the classes that implement one or more specified category identifiers.
EnumImplCategoriesOfClass	Retrieves an enumerator for the CATIDs implemented by the specified class.
EnumReqCategoriesOfClass	Retrieves an enumerator for the CATIDs required by the specified class.
GetCategoryDesc	Retrieves the localized description string for a specific category ID.
IsClassOfCategories	Determines whether a class implements one or more categories.

2. ICatRegister: provides methods for registering and unregistering component category information in the registry. This includes both the human-readable names of categories and the categories implemented or required by a given component or class.

   When to implement: You do not need to implement this interface. The component category manager, a system-provided COM object that can be instantiated by using CoCreateInstance, implements ICatRegister.

   When to use: The owner of a category uses this interface to register or unregister the human-readable names. The owner of a component uses this interface to add or remove categories implemented or required by this component.

   The ICatRegister interface has these methods.

Method	Description
RegisterCategories	Registers one or more component categories.
RegisterClassImplCategories	Registers the class as implementing one or more component categories.
RegisterClassReqCategories	Registers the class as requiring one or more component categories.
UnRegisterCategories	Removes the registration of one or more component categories.
UnRegisterClassImplCategories	Removes one or more implemented category identifiers from a class.
UnRegisterClassReqCategories	Removes one or more required category identifiers from a class.

Page intentionally left blank

Page intentionally left blank

9. Microsoft Interface Definition Language (MIDL)

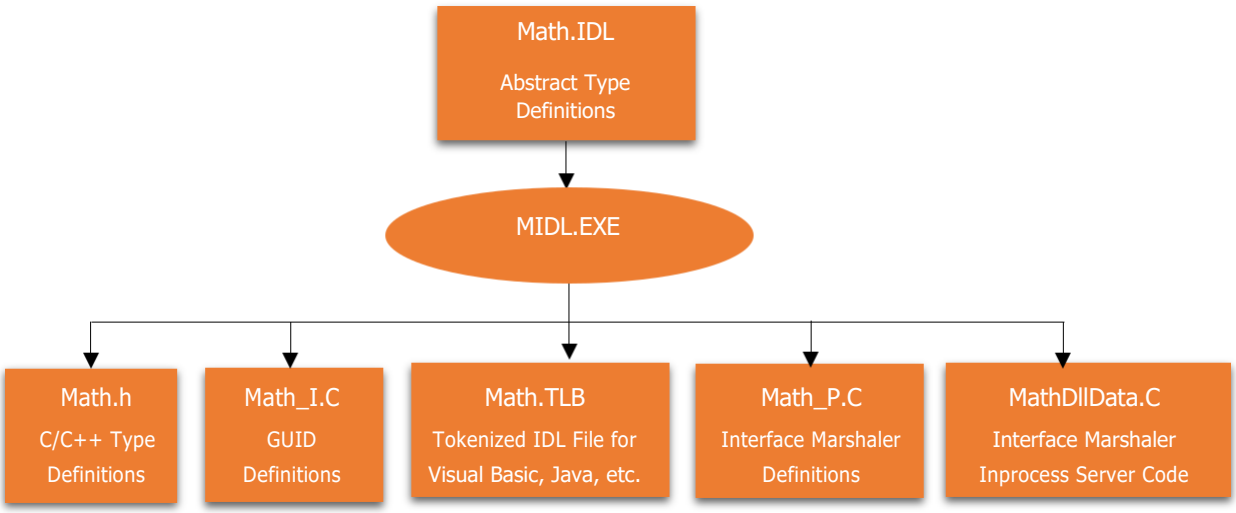
The Microsoft Interface Definition Language (MIDL) defines interfaces between client and server programs. Microsoft includes the MIDL compiler with the Platform Software Development Kit (SDK) to enable developers to create the interface definition language (IDL) files required for remote procedure call (RPC) interfaces and COM/DCOM interfaces. MIDL also supports the generation of type libraries for OLE Automation.

MIDL can be used in all client/server applications based on Windows operating systems. It can also be used to create client and server programs for heterogeneous network environments that include such operating systems as Unix and Apple. Microsoft supports the Open Group (formerly known as the Open Software Foundation) DCE standard for RPC interoperability.

9.1 IDL

COM IDL is based on the Open Software Foundation Distributed Computing Environment Remote Procedure Call (OSF DCE RPC) IDL. DCE IDL allows remote procedure calls to be described in a language-neutral manner that also enables an IDL compiler to generate networking code that transparently remotes the described operations over a variety of network transports. COM IDL simply adds a few COM specific extensions to DCE IDL to support the object oriented nature of COM (e.g. inheritance polymorphism). Not coincidentally when COM objects are accessed across execution context or machine boundries, all client-object communications use MS_RPC (an implementation of DCE RPC that is part of Windows NT/95) as the underlying transport.

The Win32 SDK includes an IDL compiler called MIDL.EXE that parses COM IDL files and generates several artifacts. As shown in below figure, MIDL generates C/C++ compatible header files that contain the abstract base class definitions that correspond to the interfaces that are defined in the original IDL file.



These header files also contain C-compatible structure based definitions that allow C programs to access or implement the IDL described interfaces. The fact that MIDL automatically generates the C/C++ header file implies that no COM interfaces should be defined manually in C++. Having a single point of definition avoids having multiple incompatible versions of an interface definition avoids having multiple incompatible versions of an interface definition that can fail out of sync over time.MIDL also generates source code that allows the interface to be used across thread, process and host boundries. Finally, MIDL can also generate a binary file that allows other COM-aware environments to produce language mapping for the interfaces defined in the original IDL file. The binary file is called a type library and contains tokenized IDL in an efficiently parsed form. Type libraries are typically distributed as part of the implementation's executable and allow languages such as Visual Baisc, Java or Object Parsal to use the interfaces that are exposed by the implementation.

```
import "<a.idl header file>"
//interface definition
[
   object,
   uuid(<string form of IID of interface>),
       helpstring("<any string>"),
       pointer_default(<either unique or ref or ptr>)
```

```
]

interface<interface name>:<base interface name>
{
   // only custom function prototypes
   HRESULT <function name>([in or out or both] data types, …);
   .
   .
};
```

Import statement: The *import* statement is similar to #include statement in C. Import statement is used to include the required idl header files. Windows SDK installation provides default IDL files for developers and available in the VC include directory. Most commonly used .idl header file is unknown.idl that provides IUnknown interface description. Most of the custom interfaces import this file using the import statement `import "unknwn.idl"`. Just like other COM interfaces, our ISum interface is also derived from the IUnknown interface. Thus, we will include *unknwn.idl* using the import statement just showed earlier.

Comment: C/C++ style comments i.e. using // and /* …*/ can be used to put comment in the idl file.

Interface definition block: this block is delimited using square brackets [] used to describe the COM object you want to export to the clients.

For custom interfaces like ISum, the first statement should be the *object* ended with a comma.

Next statement starts with *uuid* followed by a pair of parentheses used to state the string from (without quotes) of interface's IID and ends with a comma (,). Note that GUIDGEN tool generates a GUID in hex and string forms out of which the string form should be used in IDL.

The 3rd parameter is *helpstring* followed by a pair of parentheses. Put the help text for the interface in between the parentheses and end the statement with a comma (,).

The 4th and last statement is *pointer_default* followed by a pair of parentheses. Inside the parentheses mention "how the parameter pointers should be treated?" There are 3 options ref, unique and ptr. The closing bracket is not followed by comma this time.

- ref: pointers are treated by reference i.e. they will always contain a valid address, cannot be NULL and cannot be aliased.
- unique: unique pointers can be NULL. They thus can change themselves between function calls, but they cannot be aliased.
- ptr: these are equivalent to "C" pointers. They can be NULL, can change their values between in calls and can be aliased.

MIDL compiler uses above values to optimize proxy and stub dll code.

For our IAddSubtract interface, the "interface block" looks something like:

```
// ISum Interface
[
   object, uuid(8CC14612-CAB1-42dc-8E22-
   B88F5B2F0CC2),
   helpstring("ISum Interface"), /* to pass null pointer as out parameter(ptr also be used) */

   pointer_default(unique)
] //(no punctuation)
```

Interface Function Declaration Block:

First statement starts with the *interface* keyword followed by the custom interface name. After that put a colon (:) and then name of the base interface from which the custom interface is being inherited.

```
interface <custom interface name>:<base interface name>
```

After the interface statement, put a pair of braces and declare the custom interface methods prototypes in between the braces. Put a semicolon after the closing brace.

```
interface ISum : IUnknown
{
   // interface methods
   HRESULT AddInt([in] int,[in] int,[out] int*);
}; //remember the semicolon
```

9.2 DECLARE_INTERFACE & DECLARE_INTERFACE_ MACROs

When declaring and implementing our ISum COM interface manually in IDL, it will look something like following:

```
#undef INTERFACE
#define INTERFACE ISum
DECLARE_INTERFACE_(INTERFACE, IUnknown)
{
   // *** IUnknown methods ***
   STDMETHOD(QueryInterface) (THIS_ REFIID, void **)
   PURE; STDMETHOD_(ULONG, AddRef) (THIS) PURE;
   STDMETHOD_(ULONG, Release) (THIS) PURE;

   // ** ISum methods ***
   STDMETHOD_(int, AddInt)(THIS_ int, int) PURE;
};
```

Rules:

You must set the INTERFACE macro to the name of the interface being declared. Note that you need to #undef any previous value before you #define the new one.

You must use the DECLARE_INTERFACE and DECLARE_INTERFACE_ macros to generate the preliminary bookkeeping for an interface. Use DECLARE_INTERFACE for interfaces that have no base class and DECLARE_INTERFACE_ for interfaces that derive from some other interface. In our example, we derive the ISum interface from IUnknown. In practice, you will never find the plain DECLARE_INTERFACE macro because all interfaces derive from IUnknown if nothing else.

You must list all the methods of the base interfaces in exactly the same order that they are listed by that base interface; the methods that you are adding in the new interface must go last.

You must use the STDMETHOD or STDMETHOD_ macros to declare the methods. Use STDMETHOD if the return value is HRESULT andSTDMETHOD_ if the return value is some other type.

If your method has no parameters, then the argument list must be (THIS). Otherwise, you must insert THIS_ immediately after the open-parenthesis of the parameter list.

After the parameter list and before the semicolon, you must say PURE.

Inside the curly braces, you must say BEGIN_INTERFACE and END_INTERFACE.

There is a reason for each of these rules. They have to do with being able to use the same header for both C and C++ declarations and with interoperability with different compilers and platforms.

You must set the INTERFACE macro because its value is used by the THIS and THIS_ macros later.

You must use one of the DECLARE_INTERFACE* macros to ensure that the correct prologue is emitted for both C and C++. For C, a vtable structure is declared, whereas for C++ the compiler handles the vtable automatically; on the other hand, since C++ has inheritance, the macros need to specify the base class so that upcasting will work.

You must list the base class methods in exactly the same order as in the original declarations so that the C vtable structure for your derived class matches the structure for the base class for the extent that they overlap. This is required to preserve the COM rule that a derived interface can be used as a base interface.

You must use the STDMETHOD and STDMETHOD_ macros to ensure that the correct calling conventions are declared for the function prototypes. For C, the macro creates a function pointer in the vtable; for C++, the macro creates a virtual function.

The THIS and THIS_ macros are used so that the C declaration explicitly declares the "this" parameter which in C++ is implied. Different versions are needed depending on the number of parameters so that a spurious trailing comma is not generated in the zero-parameter case.

The word PURE ensures that the C++ virtual function is pure, because one of the defining characteristics of COM interfaces is that all methods are pure virtual.

The BEGIN_INTERFACE and END_INTERFACE macros emit compiler-specific goo which the compiler vendor provides in order to ensure that the generated interface matches the COM vtable layout rules.

Notice that the AddInt method takes 2 integer parameters and returns the summation of those 2 integers as an integer.

In "C" the AddInt prototype would be something like `int AddInt(int, int);` But working with IDL we need to adhere to below rules and modify it as `HRESULT AddInt([in] int,[in] int,[out] int*);`

IDL expects return types of all custom interface methods as HRESULT. Due to this, we cannot return interger instead must return HRESULT.

Output variable must of a pointer type. This is why we passed in the 3rd out parameter in the AddInt method. Client passes in the address of the output variable and gets the result value in return. In our case, client passes in NULL assigned address and in return gets summation of the integers.

Note: MIDL requires all output parameters strictly of pointer type, which applies to both [out] and [in, out] type parameters.

9.3 Attributes: [in], [out] and [in, out]

[in]: The [in] attribute indicates that a parameter is to be passed from the calling procedure to the called procedure. It is a read-only input parameter. Syntax: [in] <data type of input parameter>

In our case client has 2 read-only integers to sumup, thus passing in with [in] attribute.

[out]: The [out] attribute identifies pointer parameters that are returned from the called procedure to the calling procedure (from the server to the client).

The [out] attribute indicates that a parameter that acts as a pointer and its associated data in memory are to be passed back from the called procedure to the calling procedure.

The [out] attribute must be a pointer. DCE IDL compilers require the presence of an explicit * as a pointer declarator in the parameter declaration. Microsoft IDL offers an extension that drops this requirement and allows an array or a previously defined pointer type.

A related attribute, [in], indicates that the parameter is passed from the calling procedure to the called procedure. The [in] and [out] attributes specify the direction in which the parameters are passed. A parameter can be defined as [in]-only, [out]-only, or [in, out].

An [out]-only parameter is assumed to be undefined when the remote procedure is called and memory for the object is allocated by the server. Since top-level pointer/parameters must always point to valid storage, and therefore cannot beNULL, [out] cannot be applied to top-level [unique] or [ptr] pointers. Parameters that are [unique] or [ptr] pointers must be either [in] or [in, out] parameters.

In our case, client wants to sumup the 2 intergers and hold the result in the 3rd parameter, thus [out] attribute is used for 3rd parameter.

[in, out]: [in, out] means that a valid value is passed when the method is called and a valid value is there (where the pointer points) when the method returns success. [out] means that the value pointed to can be whatever when the method is called but it will be valid when the method returns success. Both [out] and [in, out] parameters must be pointers - their values are unchanged and valid and the validity requirements only apply to the variables they point to.

In our code 3rd parameter is purely output parameter. We can specify 3rd parameter of this syntax. When a parameter is passed with certain value to a function, in that function the passed value of this parameter is going to be used and on returning the original passed value of this parameter is going to be changed and then returned.

Note that, same syntax is applicable to integer, float, double, long, short, unsigned data-types. But, strings, arrays and structures follow different syntax. Also note that ANSI string must be converted to UNICODE string.

9.3.1 Pointers

Consider following IDL method definition:

```
HRESULT Method([in] const short *ps);

short s = 10;
HRESULT hr = p->Method(&s);
```

It would be the responsibility of the interface proxy to dereference the pointer and transmit the value 10 in the ORPC request message.

```
HRESULT hr = p->Method(0);  // pass a null pointer
```

If the calling thread executes in the apartment of the object, then there is no proxy and the null pointer will be passed directly to the object. If the object resides in a different apartment and a proxy is used, to indicate that a pointer must never be null, the interface designer can use the [ref] attribute:

```
HRESULT Method([in, ref] const short *ps); // ps cannot be a null pointer
```

Pointers that use the [ref] attribute are called *reference pointers*.

If instead, a null pointer is a legal parameter value, the IDL definition should use the [unique] attribute:

```
HRESULT Method([in, unique] const short *ps); // ps can be a null pointer
```

9.3.2 String data-type

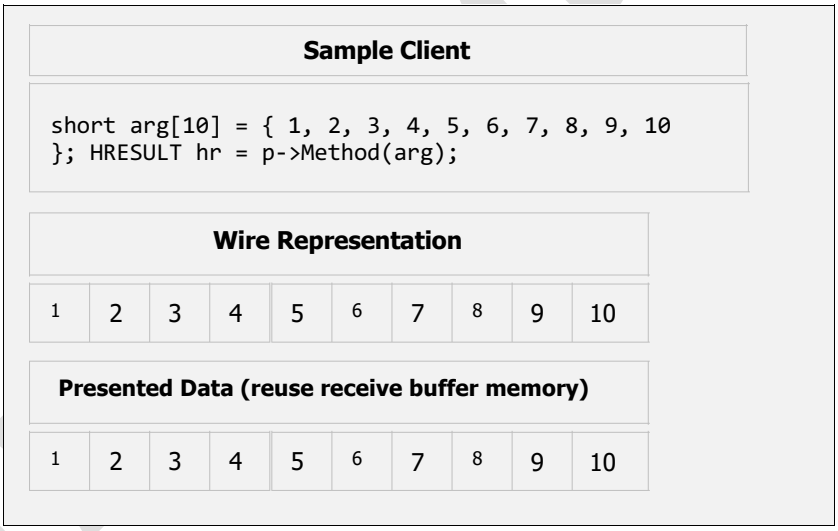
A string should be UNICODE compliant and thus must be passed either as wchar_t or OLECHAR or OLECHAR * or wchar_t * or LPOLESTR (which is nothing but OLECHAR *). These types are predefined in COM and OLE headers. This implies that, before passing in an ANSI char[] or ANSI char * to a method, first it must be converted to UNICODE format i.e. wchar_t[], wchar_t *, OLECHAR or LPOLESTR type. Similarly, char type must be converted to wchar_t or OLECHAR before sending to server.

9.3.3 Array data-type

By default, pointer parameters are assumed to be pointers to single instances, not arrays. To pass an array as a parameter, one can either use the C array syntax and/or use special IDL attributes to indicate various array dimension information. The simplest technique for passing arrays is to specify the dimentions at compile time:

HRESULT Method([in] short arg[10]);

This is formally known as a fixed array. For this array, the interface proxy will allocate 20 bytes (10 * sizeof(short)) in the ORPC request message and then copy all eight elements into the message. Once the ORPC request is received by the server, the interface stub will use the memory from the received buffer directly as an argument to the function.



To allow to be dimensioned at runtime, IDL (and the underlying wire protocol NDR) allows the caller to specify the capacity of the array at runtime. Arrays of this type are referred to as conformant arrays. IDL uses the [size_is] attribute to allow the caller to specify the conformance of an array:

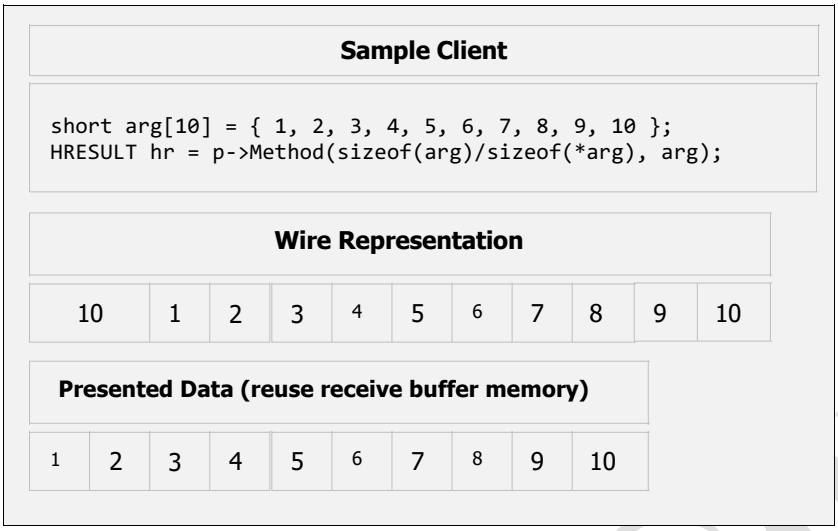
```
HRESULT Method([in] long elements, [in, size_is(elements)] short arg[*]);
```

Or

```
HRESULT Method([in] long elements, [in, size_is(elements)] short arg[]);
```

Or

```
HRESULT Method([in] long elements, [in, size_is(elements)] short *arg);
```



For array of other data-types size_is attribute is used in IDL. Use the [size_is] attribute to specify the size of memory, in elements, allocated for sized pointers, sized pointers to sized pointers, and single or multidimensional arrays.

IDL is also used for marshalling and un-marshalling of parameter data. So it is a must for both the client and the server to get the correct sized data. In case of simple/primitive data-types like int, float, double, there is no problem of marshalling and un-marshalling because these data-types have predefined size. Problem starts when an array data-type needs to be marshalled/unmarshalled. In such case, specifying the correct array size is a must. To achive this *size_is* modifier is used.

A sample method declaration in IDL that passes (one-way) character array to the server:

```
HRESULT Method1([in] unsigned short size, [in] unsigned short length,
          [size_is(size), length_is(length)] wchar_t data[*]);

Client call:

wchar_t *data = new wchar_t[10] { L"test data"};
interface_ptr->Method1(sizeof(data), wcslen(data), data, &szBSTR);

if (data != NULL) {
   delete[] data;
   *data = NULL;
}
```

A sample method declaration in IDL that passes & returns (two-way) character array to the server:

```
HRESULT Method2([in, out] short * pSize, [in, out, size_is(*pSize)] wchar_t *a[*]);

Client call:

wchar_t *data = new wchar_t[10] { L"test data"};
short size = 0;
interface_pointer->Method2(&size, data);

if (data != NULL) {
   delete[] data;
   *data = NULL;
}
```

9.3.4. Structure data-type

Sample structure and a method to use in an IDL:

```
typedef struct tagOBJSTRUCTURE
{
   int value;
   long* returnValue;

} OBJSTRUCTURE;

HRESULT Method1([in] OBJSTRUCTURE, [out]long* Count,
                    [out, size_is(, *Count)] OBJSTRUCTURE** ppStruct);
```

Server and client sides' code:

Server side code:

```
HRESULT CMyMath::Method1(OBJSTRUCTURE s1, long *Count, OBJSTRUCTURE** s2)
{
   OBJSTRUCTURE x = s1;
   *s2 = (OBJSTRUCTURE*)CoTaskMemAlloc(sizeof
   OBJSTRUCTURE); (*s2)->value = 10;
   (*s2)->returnValue = new long;
   *(*s2)->returnValue = 1000;
   *Count = 1;
   return(S_OK);
}
```

Client side code:

```
OBJSTRUCTURE s;
OBJSTRUCTURE *s1 = 0;

s.value = 20;
s.returnValue = new long;
*s.returnValue = 50;

count =  new long;
*count = 2;

s1 = new OBJSTRUCTURE();
pIMyMath->Method1(s, count, &s1);
CoTaskMemFree (s1);
```

9.3.5 Interface data-type

As every interface is of the structure type internally, using interface in IDL is similar to using the structure data-type. Here is a sample:

Interface declaration in IDL:

```
interface IMyMath : IDispatch
{
   import "oaidl.idl";
   HRESULT SumOfTwoIntegers([in]int,[in]int,[out,retval]int *);
   HRESULT SubtractionOfTwoIntegers([in]int, [in]int,[out,retval]int
   *); HRESULT Method1([in] OBJSTRUCTURE,
      [out]long* Count,
      [out, size_is(, *Count)] OBJSTRUCTURE** ppStruct);
};

HRESULT InterfaceMethod([in] IMyMath* pIMyMath,  [out] IMyMath** ppIMyMath);
```

Server and client side changes:

Server side code:

```
HRESULT CMyMath::InterfaceMethod(IMyMath *pIMyMath, IMyMath **ppIMyMath)
{
   IMyMath *x = pIMyMath;
   return QueryInterface(IID_IMyMath, (void**) ppIMyMath);
}
```

Client side code:

```
IMyMath *ppIMyMath = NULL;

hr = pIMyMath->InterfaceMethod(pIMyMath, &ppIMyMath);
ppIMyMath->Release();
ppIMyMath = NULL;
```

Some points to remember:

Always make output parameter as pointer type.
For array as output parameter use size_is() modifier to specify size of array.
String of "input" or "output" type must be either declared or converted to wchar_t or to OLECHAR or LPOLESTR type.
If interface is going to b returned as output type, then use required interface's IID with iid_is() modifier and request IUnknown to get required interface .
When a pointer is passed to any function, client should use CoTaskMemAlloc and CoTaskMemFree COM APIs, instead of compiler specific malloc, freee, new and delete APIs.
Never use void * as output parameter as idl requires specific data size for marshalling and unmarshalling. Use [out] attribute to return data from a method and return HRESULT to report method status.
Import unknwn.idl file in every IDL file.
Statements in object block must have comma at the end except the last statement of object block (usually pointer_default statement).

Page intentionally left blank

Page intentionally left blank

10. Automation Servers

Automation makes it possible for your application to manipulate objects implemented in another application, or to expose objects so they can be manipulated. An Automation server is an application that exposes programmable objects (called Automation objects) to other applications (called Automation clients). Automation servers are sometimes called Automation components.

Exposing Automation objects enables clients to automate certain procedures by directly accessing the objects and functionality the server makes available. Exposing objects this way is beneficial when applications provide functionality that is useful for other applications. For example, a word processor might expose its spell-checking functionality so that other programs can use it. Exposure of objects thus enables vendors to improve their applications' functionality by using the ready-made functionality of other applications.
These Automation objects have properties and methods as their external interface. Properties are named attributes of the Automation object. Properties are like the data members of a C++ class. Methods are functions that work on Automation objects. Methods are like the public member functions of a C++ class.

By exposing application functionality through a common, well-defined interface, Automation makes it possible to build applications in a single general programming language like Microsoft Visual Basic instead of in diverse, application-specific macro languages.

Automation in COM was first known as OLE-Automation. Automation server can serve requests coming from several clients including scripts irrespective of the language they are written in. So a component that supports automation can be accessed by clients developed in following languages:
C++
C
VB 6.0
- Visual Java
- Visual FoxPro
- VB script

Jscript
- Visual Delphi
- C# .Net

VB .Net
And many more…

This prove that COM components support language independency via automation.

NOTE: - Though Java client can use COM automation component, it requires CORBA (Common Object Request Broker Architecture) middleware to communicate across the network. Visual java takes care of CORBA middleware and therefore designing visual java client is much easier than the standard Java client. The latest Microsoft developing platform .Net and its languages like C#, VB can also access COM components using interop middlewares - RCW (Runtime Callable Wrapper) and CCW (COM Callable Wrapper).

Automation and interoperability is possible due to the fact that clients get component definitions in their own understandable format via the type library (.tlb). Type library exposes component and allows both standard and IDispatch marshalling.

The VTable is "early bound" and this therefore faster. You know the type of method you are calling at compile time. Using the IDispatch interface is "late bound" and is therefore slower, but more flexible. It's used extensively for scripting. The method and property types are determined at runtime. If you want create a COM which can be accessible from VBScript/JScript or from old "classic" ASP you have to implement IDispatch.

IDispatch interface

Exposes objects, methods and properties to programming tools and other applications that support Automation. COM components implement the IDispatch interface to enable access by Automation clients, such as Visual Basic.
The IDispatch interface inherits from the IUnknown interface.

The IDispatch interface has these methods.

Method	Description
GetIDsOfNames	Maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs, which can be used on subsequent calls to Invoke.
GetTypeInfo	Retrieves the type information for an object, which can then be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an object.

Key points to remember when developing and deploying an automation component (dll/exe):

it must not break the existing class factory client
must support Idispatch
support automation (IDispatch) and vtable bindings (IClassFactory) in IDL/type library and implement IDispatch and IClassFactory interfaces
mark automation supporting interface with dual and oleautomation attributes. The dual attribute on interface creates an interface that is both a dispatch interface and a COM interface. Because it is derived from IDispatch, a dual interface supports Automation, which is what the oleautomation attribute specifies. Interface imports Oaidl.idl to get the definition of IDispatch.
usually automation components are self executable (exe) and provide user interface
Deploy automation dll/exe and proxy middleware inside the same directory specified. In production it is usually *winnt|system32*
Keep the type library and client inside the same directory if type library is not embedded into the executable
register proxy dll and automation dll using regsvr32 command. If you automation server is an exe, use '<server exe>/regserver' command

A sample automation server (DLL) and client applications:

Applications structure			
1	AutomationServer (DLL)	Automation COM DLL server exporting COM Component	1. AutomationServer.h (header) 2. AutomationServer.cpp (source) 3. AutomationServer.def (exported functions)
2	AutomationProxyStub (DLL)	Proxy and Stub DLL for remoting (RPC) and data marshalling	1. AutomationProxyStubHeader.h (header) 2. AutomationProxyStub.c (source) 3. AutomationProxyStubDllData.c (source) 4. AutomationProxyStubGuids.c (source) 5. AutomationServerProxyStub.def (exported functions) 6. AutomationServerTypeLib.idl (Interface definition)
3	IClassFactoryAutomationClient (self-executable)	Automation client for AutomationServer component using ClassFactory i.e. CoCreateInstance	1. AutomationServer.h (header of AutomationServer.dll server) 2. IClassFactoryAutomationClient.cpp (client source)
4	IDispatchAutomationClient (self-executable)	Automation client for AutomationServer component using IDispatch interface	1. AutomationServer.h (header of AutomationServer.dll server) 2. IDispatchAutomationClient.cpp (client source)
5	CSharpAutomation (self-executable)	C# automation client for AutomationServer component using .net interop services	1. CSharpAutomation.cs (source)

6	VBAutomation (self-executable)	VB automation client for AutomationServer component using .net interop services	1. VBAutomation.cs (source)

10.1 AutomationServer (DLL)

Steps to create the server DLL project:

Step-1: Open Visual Studio 2015 IDE, go to File -> New -> Project option
Step-2: From 'Templates' section, select the Visual C++ option.
Step-3: From 'Visual C++' option, select 'Win32 Project'
Step-4: Follow the 'Win32 Application Wizard' and select following options:
- d. Application type: DLL
- e. Additional options: Empty project
- f. Do not 'Add common header files'

Step-5: Click Finish

- a. AutomationServer.h (header)
- b. AutomationServer.cpp (source)
- c. AutomationServer.def (exported functions)

Step-7: Select Release configuration and 'x64' platform
Step-8: Build the project. It should create 64-bit AutomationServer.dll. Register AutomationServer.dll using regsvr32.exe tool in the elevated mode as seen before.

AutomationServer.h (header)

```
#pragma once

class IMyMath : public IDispatch
{
public:
	// pure virtual
	virtual HRESULT __stdcall SumOfTwoIntegers(int, int, int *) = 0;

	// pure virtual
	virtual HRESULT __stdcall SubtractionOfTwoIntegers(int, int, int *) = 0;
};

// CLSID of MyMath Component : {71169462-E879-4B9B-ACFF-FB718082D292}
const CLSID CLSID_MyMath = { 0x71169462, 0xe879, 0x4b9b, 0xac, 0xff, 0xfb, 0x71, 0x80,
0x82, 0xd2, 0x92 };

// IID of ISum Interface : {2778214B-DE94-4BD8-B40B-DCF950D2FACC}
const IID IID_IMyMath = { 0x2778214b, 0xde94, 0x4bd8, 0xb4, 0xb, 0xdc, 0xf9, 0x50, 0xd2,
0xfa, 0xcc };
```

AutomationServer.cpp (source)

```
// Header Files
#include <windows.h>
#include <stdio.h>     // for swprintf_s()
#include "AutomationServer.h"

// coclass class declaration
class CMyMath :public IMyMath
{
private:
   long m_cRef;
   ITypeInfo *m_pITypeInfo =
NULL; public:
   // constructor method
   declarations CMyMath(void);
```

```
   // destructor method
   declarations ~CMyMath(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // IDispatch specific method declarations (inherited)
   HRESULT __stdcall GetTypeInfoCount(UINT*);
   HRESULT __stdcall GetTypeInfo(UINT, LCID, ITypeInfo**);
   HRESULT __stdcall GetIDsOfNames(REFIID, LPOLESTR*, UINT, LCID, DISPID*);
      HRESULT __stdcall Invoke(DISPID, REFIID, LCID, WORD, DISPPARAMS*,
      VARIANT*, EXCEPINFO*, UINT*);

   // ISum specific method declarations (inherited)
   HRESULT __stdcall SumOfTwoIntegers(int, int, int *);

   // ISubtract specific method declarations (inherited) HRESULT
   __stdcall SubtractionOfTwoIntegers(int, int, int *);

   // custom methods
   HRESULT InitInstance(void);
};

// class factory declaration
class CMyMathClassFactory :public IClassFactory
{
private:
   long m_cRef;
public:
   // constructor method declarations
   CMyMathClassFactory(void);
   // destructor method declarations
   ~CMyMathClassFactory(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // IClassFactory specific method declarations (inherited)
   HRESULT __stdcall CreateInstance(IUnknown *, REFIID, void **);
   HRESULT __stdcall LockServer(BOOL);
};

// global DLL handle
HMODULE ghModule = NULL;

// global variable declarations
long glNumberOfActiveComponents = 0;  // number of active components
long glNumberOfServerLocks = 0;      // number of locks on this dll
                                     // {1F879C17-26BE-420C-A3F6-2995E0970AF3}

const GUID LIBID_AutomationServer = {
0x5C854F26,0x4AF7,0x458E,0x8A,0x47,0x06,0x3D,0xD9,0x33,0xF2,0x65 };

// DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
   switch (dwReason)
   {
   case DLL_PROCESS_ATTACH:
      ghModule = hDll;
      break;
   case DLL_PROCESS_DETACH:
```

```
      break;
   }
   return(TRUE);
}

// Implementation Of CMyMath's Constructor Method
CMyMath::CMyMath(void)
{
      m_cRef = 1; // hardcoded initialization to anticipate possible failure of
                  // QueryInterface()
   InterlockedIncrement(&glNumberOfActiveComponents); // increment global counter
}

// Implementation Of CMyMath's Destructor Method
CMyMath::~CMyMath(void)
{
   InterlockedDecrement(&glNumberOfActiveComponents); // decrement global counter
}

// Implementation Of CMyMath's IUnknown's Methods
HRESULT CMyMath::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<IMyMath *>(this);
   else if (riid == IID_IDispatch)
      *ppv = static_cast<IMyMath *>(this);
   else if (riid == IID_IMyMath)
      *ppv = static_cast<IMyMath *>(this);
   else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

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

ULONG CMyMath::AddRef(void)
{
   InterlockedIncrement(&m_cRef);
   return(m_cRef);
}

ULONG CMyMath::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      m_pITypeInfo->Release();
      m_pITypeInfo = NULL;
      delete(this);
      return(0);
   }

   return(m_cRef);
}

// Implementation Of IMyMath's Methods
HRESULT CMyMath::SumOfTwoIntegers(int num1, int num2, int *pSum)
{
   *pSum = num1 + num2;
   return(S_OK);
}
```

```
HRESULT CMyMath::SubtractionOfTwoIntegers(int num1, int num2, int
*pSubtract) {
   *pSubtract = num1 - num2;
   return(S_OK);
}

HRESULT CMyMath::InitInstance(void)
{
   // function declarations
   void ComErrorDescriptionString(HWND, HRESULT);

   // variable declarations
   HRESULT hr;
   ITypeLib *pITypeLib = NULL;

   // code
   if (m_pITypeInfo == NULL)
   {
      hr = LoadRegTypeLib(LIBID_AutomationServer,
         1, 0,// major/minor version numbers
         0x00,// LANG_NEUTRAL
         &pITypeLib);

      if (FAILED(hr))
      {
         ComErrorDescriptionString(NULL, hr);
         return(hr);
      }

      hr = pITypeLib->GetTypeInfoOfGuid(IID_IMyMath, &m_pITypeInfo);

      if (FAILED(hr))
      {
         ComErrorDescriptionString(NULL, hr);
         pITypeLib->Release();
         return(hr);
      }

      pITypeLib->Release();
   }

   return(S_OK);
}

// Implementation Of CMyMathClassFactory's Constructor Method
CMyMathClassFactory::CMyMathClassFactory(void)
{
   m_cRef = 1; // hardcoded initialization to anticipate possible failure of
               // QueryInterface()
}

// Implementation Of CMyMathClassFactory's Destructor Method
CMyMathClassFactory::~CMyMathClassFactory(void)
{
   // code (nothing for us)
}

// Implementation Of CMyMathClassFactory's IClassFactory's IUnknown's Methods
HRESULT CMyMathClassFactory::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<IClassFactory *>(this);
   else if (riid == IID_IClassFactory)
      *ppv = static_cast<IClassFactory
   *>(this); else
   {
```

```
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

   reinterpret_cast<IUnknown *>(*ppv)->AddRef();

   return(S_OK);
}

ULONG CMyMathClassFactory::AddRef(void)
{
   InterlockedIncrement(&m_cRef);

   return(m_cRef);
}

ULONG CMyMathClassFactory::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);

      return(0);
   }
   return(m_cRef);
}

// Implementation Of CMyMathClassFactory's IClassFactory's Methods
HRESULT CMyMathClassFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid, void **ppv)
{
   // variable declarations
   CMyMath *pCMyMath = NULL;
   HRESULT hr;

   if (pUnkOuter != NULL)
      return(CLASS_E_NOAGGREGATION);

   // create the instance of component i.e. of CMyMath class
   pCMyMath = new CMyMath;
   if (pCMyMath == NULL)
      return(E_OUTOFMEMORY);

   // call automation related init method
   pCMyMath->InitInstance();

   // get the requested interface
   hr = pCMyMath->QueryInterface(riid, ppv);

   pCMyMath->Release();// anticipate possible failure of QueryInterface()

   return(hr);
}

HRESULT CMyMathClassFactory::LockServer(BOOL fLock)
{
   if (fLock)
      InterlockedIncrement(&glNumberOfServerLocks);
   else
      InterlockedDecrement(&glNumberOfServerLocks);

   return(S_OK);
}
```

```
// Implementation of IDispatch's methods
HRESULT CMyMath::GetTypeInfoCount(UINT *pCountTypeInfo)
{
   // as we have type library it is 1, else 0
   *pCountTypeInfo = 1;

   return(S_OK);
}

HRESULT CMyMath::GetTypeInfo(UINT iTypeInfo, LCID lcid, ITypeInfo **ppITypeInfo)
{
   *ppITypeInfo = NULL;

   if (iTypeInfo != 0)
      return(DISP_E_BADINDEX);

   m_pITypeInfo->AddRef();

   *ppITypeInfo = m_pITypeInfo;

   return(S_OK);
}

HRESULT CMyMath::GetIDsOfNames(REFIID riid, LPOLESTR *rgszNames, UINT cNames, LCID lcid,
DISPID *rgDispId)
{
   return(DispGetIDsOfNames(m_pITypeInfo, rgszNames, cNames, rgDispId));
}

HRESULT CMyMath::Invoke(DISPID dispIdMember, REFIID riid, LCID lcid, WORD wFlags,
DISPPARAMS *pDispParams, VARIANT *pVarResult, EXCEPINFO *pExcepInfo, UINT *puArgErr)
{
   // variable declarations
   HRESULT hr;

   hr = DispInvoke(this,
      m_pITypeInfo,
      dispIdMember,
      wFlags,
      pDispParams,
      pVarResult,
      pExcepInfo,
      puArgErr);

   return(hr);
}

// Implementation Of Exported Functions From This Dll
extern "C" HRESULT __stdcall DllGetClassObject(REFCLSID rclsid, REFIID riid, void **ppv)
{
   // variable declaraions
   CMyMathClassFactory *pCMyMathClassFactory = NULL;
   HRESULT hr;

   if (rclsid != CLSID_MyMath)
      return(CLASS_E_CLASSNOTAVAILABLE);

   // create class factory
   pCMyMathClassFactory = new CMyMathClassFactory;

   if (pCMyMathClassFactory == NULL)
      return(E_OUTOFMEMORY);

   hr = pCMyMathClassFactory->QueryInterface(riid, ppv);

   pCMyMathClassFactory->Release();// anticipate possible failure of QueryInterface()
```

```
   return(hr);
}

extern "C" HRESULT __stdcall DllCanUnloadNow(void)
{
   if ((glNumberOfActiveComponents == 0) && (glNumberOfServerLocks == 0))
      return(S_OK);
   else
      return(S_FALSE);
}

void ComErrorDescriptionString(HWND hwnd, HRESULT hr)
{
   // variable declarations
   TCHAR* szErrorMessage = NULL;
   TCHAR str[255];

   if (FACILITY_WINDOWS == HRESULT_FACILITY(hr))
      hr = HRESULT_CODE(hr);

      if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
      NULL, hr, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&szErrorMessage, 0,
      NULL) != 0)
   {
      swprintf_s(str, TEXT("%#x : %s"), hr, szErrorMessage);
      LocalFree(szErrorMessage);
   }
   else
            swprintf_s(str, TEXT("[Could not find a description for error # %#x.]\n"),
            hr);

   MessageBox(hwnd, str, TEXT("COM Error"), MB_OK);
}

// Registers DLL with COM into system registry
STDAPI DllRegisterServer()
{
   HKEY hCLSIDKey = NULL, hInProcSvrKey = NULL;
   LONG lRet;
   TCHAR szModulePath[MAX_PATH];
   TCHAR szClassDescription[] = TEXT("Automation COM class");
   TCHAR szThreadingModel[] = TEXT("Apartment");

   __try
   {
      // Create a key under CLSID for our COM server.
      // 32bit server on 32bit OS -> HKEY_CLASSES_ROOT\CLSID\{71169462-E879-4B9B-
      // ACFF-FB718082D292}
      // 32bit server on 64bit OS ->
      // HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{71169462-E879-4B9B-ACFF-
      // FB718082D292}
      // 64bit server on 32bit OS -> Invalid
      // 64bit server on 64bit OS -> HKEY_CLASSES_ROOT\CLSID\{71169462-E879-4B9B-
      // ACFF-FB718082D292}

      lRet = RegCreateKeyEx(HKEY_CLASSES_ROOT, TEXT("CLSID\\{71169462-E879-4B9B-ACFF-
      FB718082D292}"), 0, NULL, REG_OPTION_NON_VOLATILE, KEY_SET_VALUE |
      KEY_CREATE_SUB_KEY, NULL, &hCLSIDKey, NULL);

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

            // The default value of the key is a human-readable description of the
            // coclass.
```

```
      lRet = RegSetValueEx(hCLSIDKey, NULL, 0, REG_SZ, (const BYTE*)szClassDescription,
      sizeof(szClassDescription));

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

      // Create the InProcServer32 key, which holds info about our coclass.

      lRet = RegCreateKeyEx(hCLSIDKey, TEXT("InProcServer32"), 0, NULL,
      REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hInProcSvrKey, NULL);

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

      // The default value of the InProcServer32 key holds the full path to our DLL.

      GetModuleFileName(ghModule, szModulePath, MAX_PATH);

      lRet = RegSetValueEx(hInProcSvrKey, NULL, 0, REG_SZ, (const BYTE*)szModulePath,
      sizeof(TCHAR) * (lstrlen(szModulePath) + 1));

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

      // The ThreadingModel value tells COM how it should handle threads in our DLL.

      lRet = RegSetValueEx(hInProcSvrKey, TEXT("ThreadingModel"), 0,
         REG_SZ, (const BYTE*)szThreadingModel, sizeof(szThreadingModel));

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);
   }
   __finally
   {
      if (NULL != hCLSIDKey)
         RegCloseKey(hCLSIDKey);

      if (NULL != hInProcSvrKey)
         RegCloseKey(hInProcSvrKey);
   }

   return S_OK;
}

// Unregisters DLL from system registry
STDAPI DllUnregisterServer()
{
      RegDeleteKey(HKEY_CLASSES_ROOT, TEXT("CLSID\\{71169462-E879-4B9B-ACFF-
      FB718082D292}\\InProcServer32"));

      RegDeleteKey(HKEY_CLASSES_ROOT, TEXT("CLSID\\{71169462-E879-4B9B-ACFF-
      FB718082D292}"));

      return S_OK;
}
```

AutomationServer.def (exported functions)

```
LIBRARY AutomationServer
EXPORTS
    DllRegisterServer   PRIVATE
    DllUnregisterServer PRIVATE
    DllGetClassObject   PRIVATE
    DllCanUnloadNow     PRIVATE
```

10.2 AutomationProxyStub (DLL)

1. Open the 'Developer Command Prompt for VS2015' and run following command

```
midl /env x64 /h AutomationProxyStubHeader.h /iid AutomationProxyStubGuids.c /dlldata
AutomationProxyStubDllData.c /proxy AutomationProxyStub.c AutomationServerTypeLib.idl
```

2. This command should create 64-bit compliant prox/stub (RPC) code as follows
    a. AutomationProxyStubHeader.h (header)
    b. AutomationProxyStub.c (source)
    c. AutomationProxyStubDllData.c (source)
    d. AutomationProxyStubGuids.c (source)
    e. AutomationServerTypeLib.tlb (type library)
3. Now create proxy/stub dll for remoting and marshalling purpose:
   Steps to create the server DLL project:

   Step-1: Open Visual Studio 2015 IDE, go to File -> New -> Project option
   Step-2: From 'Templates' section, select the Visual C++ option.
   Step-3: From 'Visual C++' option, select 'Win32 Project'
   Step-4: Follow the 'Win32 Application Wizard' and select following options:
    a. Application type: DLL
    b. Additional options: Empty project
    c. Do not 'Add common header files'

    a. AutomationProxyStubHeader.h (header)
    b. AutomationProxyStub.c (source)
    c. AutomationProxyStubDllData.c (source)
    d. AutomationProxyStubGuids.c (source)
    e. AutomationServerProxyStub.def (source)

   Step-7: Select Release configuration and 'x64' platform
   Step-8: Now add reference (explicitly for each configuration and platform) of the **rpcrt4.lib** library file under the "Project Properties -> Configuration Properties -> Linker -> Input -> Additional Dependencies" option.
   Step-9: Add REGISTER_PROXY_DLL MACRO (explicitly for each configuration and platform) under "Project Properties -> Configuration Properties -> C/C++ -> Preprocessor -> Preprocessor Definitions" option. By defining the REGISTER_PROXY_DLL macro, while compiling Dlldata.c file, your proxy/stub marshaling DLL will automatically include default definitions for the DllMain, DllRegisterServer, and DllUnregisterServer functions.
   You can use these functions to self-register your proxy DLL in the system registry.
   Step-10: Build the project. It should create 64-bit AutomationProxyStub.dll.
   Step-11: Register type library (.tlb) file using the following command

```
C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\regtlibv12.exe AutomationServerTypeLib.tlb
```

Note: regtlibv12.exe is not available on Windows-10 system, but you can copy it from C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319 folder from other systems to register the tlb file.

AutomationServerProxyStub.def

```
LIBRARY AutomationProxyStub.dll
EXPORTS
      DllGetClassObject   PRIVATE
      DllCanUnloadNow     PRIVATE
      GetProxyDllInfo     PRIVATE
      DllRegisterServer   PRIVATE
      DllUnregisterServer PRIVATE
```

AutomationServerTypeLib.idl

```
import "unknwn.idl" ;
// IMyMath Interface
[
    object,
```

```
    uuid(2778214B-DE94-4BD8-B40B-DCF950D2FACC),// IID Of IMyMath
    helpstring("IMyMath Interface"),
    pointer_default(unique),
    dual,
    oleautomation
]
interface IMyMath : IDispatch
{
   import "oaidl.idl";
       HRESULT SumOfTwoIntegers([in] int, [in] int, [out, retval] int*);
   HRESULT SubtractionOfTwoIntegers([in] int, [in] int, [out, retval] int*);
};
// The Actual TypeLib Related Code
[
    uuid(5C854F26-4AF7-458E-8A47-063DD933F265),// LIBID Of Type
    Library version(1.0),// major version number.minor version number
    helpstring("MyMath Component's Type Library")
]
library AutomationServerTypeLib
{
    importlib("stdole32.tlb");
    // component code
    [
        uuid(71169462-E879-4B9B-ACFF-FB718082D292), // CLSID Of MyMath
        Component helpstring("Mat Automation Component Class")
    ]
    coclass CMyMath
    {
        [default]interface IMyMath;
    };
};
```

10.3 IClassFactoryAutomationClient (self-executable)

Steps to create the client exe project:

Step-1: Open Visual Studio IDE, go to File -> New -> Project option
Step-2: From 'Templates' section, select the Visual C++ option.
Step-3: From 'Visual C++' option, select 'Win32 Project'
Step-4: Follow the 'Win32 Application Wizard' and select following options:

d. Application type: Windows application
e. Additional options: Empty project
f. Do not 'Add common header files'

a. AutomationServer.h
b. IClassFactoryAutomationClient.cpp

Step-7: Select Release configuration and 'x64' platform
Step-8: Build the project. It should create 64-bit IClassFactoryAutomationClient.exe

AutomationServer.h: consume server's header file

IClassFactoryAutomationClient.cpp

```
#include<windows.h>
#include <stdio.h> // for swprintf_s()
#include"AutomationServer.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// global variable declarations
IMyMath *pIMyMath = NULL;
```

```
// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmdLine, int nCmdShow)
{
   // variable declarations
   WNDCLASSEX wndclass;
   HWND hwnd;
   MSG msg;
   TCHAR AppName[] =
   TEXT("ComClient"); HRESULT hr;

   // COM Initialization
   hr = CoInitialize(NULL);

   if (FAILED(hr))
   {
            MessageBox(NULL, TEXT("COM Library Can Not Be Initialized.\nProgram Will
            Now Exit."), TEXT("Program Error"), MB_OK);
      exit(0);
   }

   // WNDCLASSEX initialization wndclass.cbSize =
   sizeof(wndclass); wndclass.style = CS_HREDRAW | CS_VREDRAW;
   wndclass.cbClsExtra = 0; wndclass.cbWndExtra = 0;
   wndclass.lpfnWndProc = WndProc; wndclass.hIcon =
   LoadIcon(NULL, IDI_APPLICATION); wndclass.hCursor =
   LoadCursor(NULL, IDC_ARROW); wndclass.hbrBackground =
   (HBRUSH)GetStockObject(WHITE_BRUSH); wndclass.hInstance =
   hInstance;


   wndclass.lpszClassName = AppName;
   wndclass.lpszMenuName = NULL;
   wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

   // register window class
   RegisterClassEx(&wndclass);

   // create window
   hwnd = CreateWindow(AppName,
      TEXT("Client Of COM Dll Server"),
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL);

   ShowWindow(hwnd, nCmdShow);

   UpdateWindow(hwnd);

   // message loop
   while (GetMessage(&msg, NULL, 0, 0))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   // COM Un-initialization
   CoUninitialize();
   return((int)msg.wParam);
}
```

```
// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
   // function declarations
   void ComErrorDescriptionString(HWND, HRESULT);
   void SafeInterfaceRelease(void);

   // variable declarations
   HRESULT hr;
   int iNum1, iNum2, iSum,
   iSubtract; TCHAR str[255];

   switch (iMsg)
   {
   case WM_CREATE:
      hr = CoCreateInstance(CLSID_MyMath, NULL,
         CLSCTX_INPROC_SERVER, IID_IMyMath, (void **)&pIMyMath);

      if (FAILED(hr))
      {
         ComErrorDescriptionString(hwnd, hr);
         DestroyWindow(hwnd);
      }

      // initialize arguments hardcoded
      iNum1 = 155;
      iNum2 = 145;

      // call SumOfTwoIntegers() of IMyMath to get the sum
      pIMyMath->SumOfTwoIntegers(iNum1, iNum2, &iSum);
      wsprintf(str, TEXT("Sum Of %d And %d Is %d"), iNum1, iNum2, iSum);
      MessageBox(hwnd, str, TEXT("SumOfTwoIntegers"), MB_OK);

      // call SumOfTwoIntegers() of IMyMath to get the sum
      pIMyMath->SubtractionOfTwoIntegers(iNum1, iNum2, &iSubtract);
               wsprintf(str, TEXT("Subtraction Of %d And %d Is %d"), iNum1, iNum2,
               iSubtract);
      MessageBox(hwnd, str, TEXT("SubtractionOfTwoIntegers"), MB_OK);

      // release
      pIMyMath->Release();
      pIMyMath = NULL;// make relesed interface NULL
                  // exit the application
      DestroyWindow(hwnd);
      break;
   case WM_DESTROY:
      SafeInterfaceRelease();
      PostQuitMessage(0);
      break;
   }

   return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}

void ComErrorDescriptionString(HWND hwnd, HRESULT hr)
{
   // variable declarations
   TCHAR* szErrorMessage = NULL;
   TCHAR str[255];

   if (FACILITY_WINDOWS == HRESULT_FACILITY(hr))
      hr = HRESULT_CODE(hr);

       if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
       NULL, hr, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&szErrorMessage, 0,
       NULL) != 0)
```

```
    {
        swprintf_s(str, TEXT("%s"), szErrorMessage);
        LocalFree(szErrorMessage);
    }
    else
        swprintf_s(str, TEXT("[Could not find a description for error # %#x.]\n"), hr);

    MessageBox(hwnd, str, TEXT("COM Error"), MB_OK);
}

void SafeInterfaceRelease(void)
{
    if (pIMyMath)
    {
        pIMyMath->Release();
        pIMyMath = NULL;
    }
}
```

10.4 IDispatchAutomationClient (self-executable)

Refer to the IClassFactoryAutomationClient project's creation steps to create IDispatchAutomationClient client project.

AutomationServer.h: consume server's header file

IDispatchAutomationClient.cpp

```
#include<windows.h>
#include <stdio.h> // for swprintf_s()
#include"AutomationServer.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    // variable declarations
    WNDCLASSEX wndclass;
    HWND hwnd;
    MSG msg;
    TCHAR AppName[] = TEXT("Client");

    wndclass.cbSize = sizeof(wndclass);
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.lpfnWndProc = WndProc;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.hInstance = hInstance;
    wndclass.lpszClassName = AppName;
    wndclass.lpszMenuName = NULL;
    wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // register window class
    RegisterClassEx(&wndclass);

    // create window
    hwnd = CreateWindow(AppName,
        TEXT("Client Of Exe Server"),
        WS_OVERLAPPEDWINDOW,
```

```
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL);

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return((int)msg.wParam);
}

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    // function declarations
    void ComErrorDescriptionString(HWND, HRESULT);

    // variable declarations
    IDispatch *pIDispatch = NULL;
    HRESULT hr;
    DISPID dispid;
    OLECHAR *szFunctionName1 = L"SumOfTwoIntegers";
    OLECHAR *szFunctionName2 =
    L"SubtractionOfTwoIntegers"; VARIANT vArg[2], vRet;
    DISPPARAMS param = { vArg,0,2,NULL
    }; int n1, n2;
    TCHAR str[255];

    switch (iMsg)
    {
    case WM_CREATE:
        // initialize COM library
        hr = CoInitialize(NULL);

        if (FAILED(hr))
        {
            ComErrorDescriptionString(hwnd, hr);
            MessageBox(hwnd, TEXT("COM library can not be initialized"), TEXT("COM Error"),
            MB_OK);

            DestroyWindow(hwnd);
            exit(0);
        }

        // get ISum Interface
        hr = CoCreateInstance(CLSID_MyMath,
            NULL,
            CLSCTX_INPROC_SERVER,
            IID_IDispatch,
            (void **)&pIDispatch);

        if (FAILED(hr))
        {
            ComErrorDescriptionString(hwnd, hr);
```

```
    MessageBox(hwnd, TEXT("Component Can Not Be Created"), TEXT("COM Error"), MB_OK |
    MB_ICONERROR | MB_TOPMOST);

    DestroyWindow(hwnd);
    exit(0);
}

        // *** common code for both IMyMath->SumOfTwoIntegers() and IMyMath-
        // >SubtractionOfTwoIntegers() ***
n1 = 75;
n2 = 25;
// as DISPPARAMS rgvarg member receives parameters in reverse order
VariantInit(vArg);
vArg[0].vt = VT_INT;
vArg[0].intVal = n2; vArg[1].vt
= VT_INT; vArg[1].intVal = n1;
param.cArgs = 2;
param.cNamedArgs = 0;
param.rgdispidNamedArgs = NULL;

// reverse order of parameters
param.rgvarg = vArg;
// return value

VariantInit(&vRet);

// *** code for IMyMath->SumOfTwoIntegers() ***
hr = pIDispatch->GetIDsOfNames(IID_NULL,
    &szFunctionName1,
    1,
    GetUserDefaultLCID(),
    &dispid);

if (FAILED(hr))
{
    ComErrorDescriptionString(hwnd, hr);
    MessageBox(NULL, TEXT("Can Not Get ID For SumOfTwoIntegers()"), TEXT("Error"),
    MB_OK | MB_ICONERROR | MB_TOPMOST);
    pIDispatch->Release();

    DestroyWindow(hwnd);
}

hr = pIDispatch->Invoke(dispid,
    IID_NULL,
    GetUserDefaultLCID(),
    DISPATCH_METHOD,
    &param,
    &vRet,
    NULL,
    NULL);

if (FAILED(hr))
{
    ComErrorDescriptionString(hwnd, hr);
    MessageBox(NULL, TEXT("Can Not Invoke Function"), TEXT("Error"), MB_OK |
    MB_ICONERROR | MB_TOPMOST);
    pIDispatch->Release();

    DestroyWindow(hwnd);
}
else
{
    wsprintf(str, TEXT("Sum Of %d And %d Is %d"), n1, n2, vRet.lVal);
    MessageBox(hwnd, str, TEXT("SumOfTwoIntegers"), MB_OK);
}
```

```
        // *** code for IMyMath->SubtractionOfTwoIntegers() ***
        hr = pIDispatch->GetIDsOfNames(IID_NULL,
            &szFunctionName2,
            1,
            GetUserDefaultLCID(),
            &dispid);

        if (FAILED(hr))
        {
            ComErrorDescriptionString(hwnd, hr);
            MessageBox(NULL, TEXT("Can Not Get ID For
            SubtractionOfTwoIntegers()"), TEXT("Error"), MB_OK | MB_ICONERROR |
            MB_TOPMOST); pIDispatch->Release();

            DestroyWindow(hwnd);
        }

        // Invoke() for IMyMath->SubtractionOfTwoIntegers()
        hr = pIDispatch->Invoke(dispid,
            IID_NULL,
            GetUserDefaultLCID(),
            DISPATCH_METHOD,
            &param,
            &vRet,
            NULL,
            NULL);

        if (FAILED(hr))
        {
            ComErrorDescriptionString(hwnd, hr);
            MessageBox(NULL, TEXT("Can Not Invoke Function"), TEXT("Error"), MB_OK |
            MB_ICONERROR | MB_TOPMOST);
            pIDispatch->Release();

            DestroyWindow(hwnd);
        }
        else
        {
            wsprintf(str, TEXT("Subtraction Of %d And %d Is %d"), n1, n2, vRet.lVal);
            MessageBox(hwnd, str, TEXT("SubtractionOfTwoIntegers"), MB_OK);
        }

        // clean-up
        VariantClear(vArg);
        VariantClear(&vRet);
        pIDispatch->Release();
        pIDispatch = NULL;
        DestroyWindow(hwnd);
        break;
    case WM_DESTROY:
        CoUninitialize();
        PostQuitMessage(0);
        break;
    }

    return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}

void ComErrorDescriptionString(HWND hwnd, HRESULT hr)
{
    // variable declarations
    TCHAR* szErrorMessage = NULL;
    TCHAR str[255];

    if (FACILITY_WINDOWS == HRESULT_FACILITY(hr))
```

```
        hr = HRESULT_CODE(hr);

          if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
          NULL, hr, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&szErrorMessage, 0,
          NULL) != 0)
    {
       swprintf_s(str, TEXT("%s"), szErrorMessage);
       LocalFree(szErrorMessage);
    }
    else
       swprintf_s(str, TEXT("[Could not find a description for error # %#x.]\n"), hr);

    MessageBox(hwnd, str, TEXT("COM Error"), MB_OK);
}
```

10.5 CSharpAutomation (self-executable)

Assuming 64 Bit COM Automation Dll Server, COM Proxy/Stub Dll And TypeLibrary Tlb Are Created And Copied To Windows\System32 And From Here The Proxy/Stub Dll Is Already Registered By RegSvr32 Command.
Then From Windows\System32\Hosted Tlb File, A DotNet Compliant Dll Is Created Into The C# Source Code directory By Using Following Command In Visual Studio 2015's "64 Bit Native Command Prompt" :

```
tlbimp c:\Windows\System32\AutomationServerTypeLib.tlb
/out:.\AutomationServerTypeLibForDotNet.dll
```

The Resulting DotNet Compliant Dll Is Used In C# Code As Namespace And As Well As On Compiler Command Line

To Create The Executable:

```
csc.exe /r:AutomationServerTypeLibForDotNet.dll CSharpAutomation.cs
```

Run CSharpAutomation.exe

CSharpAutomation.cs

```
using System;
using System.Runtime.InteropServices;
using AutomationServerTypeLibForDotNet;

public class CSharpAutomation
{
   public static void Main()
   {
        CMyMathClass objCMyMathClass = new CMyMathClass();
        IMyMath objIMyMath = (IMyMath)objCMyMathClass; int
        num1=75,num2=25,sum,sub;
        sum=objIMyMath.SumOfTwoIntegers(num1, num2);
        Console.WriteLine("Sum Of " + num1 + " And " + num2 + " Is " + sum);
        sub = objIMyMath.SubtractionOfTwoIntegers(num1, num2);
        Console.WriteLine("Subtraction Of " + num1 + " And " + num2 + " Is " + sub);
    }
}
```

10.6 VBAutomation (self-executable)

Assuming 64 Bit COM Automation Dll Server, COM Proxy/Stub Dll And TypeLibrary Tlb Are Created And Copied To Windows\System32 And From Here The Proxy/Stub Dll Is Already Registered By RegSvr32 Command.

Then From Windows\System32\Hosted Tlb File, A DotNet Compliant Dll Is Created Into The VB Source Code directory By Using Following Command In Visual Studio 2013's "64 Bit Native Command Prompt":

```
tlbimp c:\Windows\System32\AutomationServerTypeLib.tlb
/out:.\AutomationServerTypeLibForDotNet.dll
```

The Resulting DotNet Compliant Dll Is Used In VB Code As Namespace And As Well As On Compiler Command Line

To Create The Executable:

```
vbc /t:winexe /r:Microsoft.VisualBasic.dll
/r:AutomationServerTypeLibForDotNet.dll VBAutomation.vb
```

Run VBAutomation.exe

VBAutomation.vb

```
' for 'Form'
Imports System.Windows.Forms
' .Net callable dll created by using tlbimp.exe
utility Imports AutomationServerTypeLibForDotNet

Public Class VBAutomation
    Inherits Form
    Public Sub New()
        Dim MyIDispatch As Object
        Dim MyRef As New CMyMathClass
        MyIDispatch = MyRef
        Dim iNum1 = 175
        Dim iNum2 = 125
        Dim iSum = MyIDispatch.SumOfTwoIntegers(iNum1, iNum2)
        Dim str As String = String.Format("Sum Of {0} And {1} Is {2}", iNum1, iNum2, iSum)
        ' default message box with only 1 button of
        'Ok' MsgBox(str)
        Dim iSub = MyIDispatch.SubtractionOfTwoIntegers(iNum1, iNum2)
        str = String.Format("Subtraction Of {0} And {1} Is {2}", iNum1, iNum2,
        iSub) MsgBox(str)
        ' following statement i.e. 'End' works as DestroyWindow(hwnd)
        End
    End Sub

    <STAThread()>
    Shared Sub Main()
        Application.EnableVisualStyles()
        Application.Run(New VBAutomation())
    End Sub
End Class
```

Page intentionally left blank

Page intentionally left blank

11. Monikers

In general, a moniker is a name or a nickname. In Microsoft's Component Object Model (COM) a moniker is an object (or component) that refers to a specific instance of another object. Monikers originated in Microsoft's Object Linking and Embedding (OLE technology as a means of linking objects.

A moniker may refer to any single object, or may be a composite made of a number of separate monikers, each of which refers to a particular instantiation of an object. The moniker is sometimes referred to as an "intelligent name," because it retains information about how to create, initialize , and bind to a single instance of an object. Once created, the moniker holds this information, as well as information about the object's states in that specific instantiation.

Since COM is not language-specific, a moniker can be used with any programming language. The programmer gives the instantiation of the object a name. By calling the moniker in code, a programmer can refer to the same object with the same states. If, for example, a moniker is created for a query , the programmer can reuse the query simply by calling the moniker in the code, because the moniker itself has the necessary information.

Additionally, a moniker in COM is not only a way to identify an object but also implemented as an object. This object provides services allowing a component to obtain a pointer to the object identified by the moniker. This process is referred to as binding. Monikers are objects that implement the **IMoniker** interface and are generally implemented in DLLs as component objects. There are two ways of viewing the use of monikers: as a moniker client, a component that uses a moniker to get a pointer to another object; and as a moniker provider, a component that supplies monikers identifying its objects to moniker clients.

IMoniker interface

Enables you to use a moniker object, which contains information that uniquely identifies a COM object. An object that has a pointer to the moniker object's IMoniker interface can locate, activate, and get access to the identified object without having any other specific information on where the object is actually located in a distributed system.

Monikers are used as the basis for linking in COM. A linked object contains a moniker that identifies its source. When the user activates the linked object to edit it, the moniker is bound; this loads the link source into memory.

If you decide you need to write your own implementation of IMoniker, you must also implement the **IROTData** interface on your moniker class. This interface allows your monikers to be registered with the running object table (ROT).

In COM technology moniker is known as Universal locater object. In short moniker gives you instance of already initialized and register object. It picks up the already running instance of needed object from Running Object Table (ROT), which is a part of COM's registry.

Types of Moniker:

Moniker Type	Creation Function	Purpose
File moniker	CreateFileMoniker	A file moniker acts as a wrapper for the pathname of a file.
Item moniker	CreateItemMoniker	An item moniker identifies an object contained in another object.
Pointer moniker	CreatePointerMoniker	A pointer moniker identifies an object that can exist only in the active or running state.
Anti-moniker	CreateAntiMoniker	An anti-moniker is the inverse of another moniker; when the two are combined, they obliterate each other.
Composite moniker	CreateGenericComposite	A composite moniker is composed of other monikers.
Class moniker	CreateClassMoniker	A class moniker acts as a wrapper for the CLSID of a COM class.

URL moniker	CreateURLMoniker	A URL moniker represents and manages a Uniform Resource Locator (URL).
OBJREF moniker*	CreateObjrefMoniker	An OBJREF moniker encapsulates a marshalled *IUnknown* interface pointer to an object.

The IMoniker interface has 15 methods. To ensure that all moniker objects support persistence, IMoniker is derived from the IPersistStream interface (4 methods), which is derived from the IPersist interface (1 method), which is in turn derived from the IUnknown interface (3 methods). That makes a grand total of 23 methods required to implement a custom moniker.

The **IMoniker** interface has these methods.

	IMoniker Method	Description
1	BindToObject	Binds to the object named by the moniker
2	BindToStorage	Binds to the object's storage
3	Reduce	Reduces the moniker to its simplest form
4	ComposeWith	Combines the moniker with another moniker to create a composite moniker (a collection of monikers stored in left-to-right sequence)
5	Enum	Enumerates component monikers
6	IsEqual	Compares the moniker with another moniker
7	Hash	Returns a hash value
8	IsRunning	Checks whether the object is running
9	GetTimeOfLastChange	Returns time the object was last changed
10	Inverse	Returns the inverse of the moniker
11	CommonPrefixWith	Finds the prefix that the moniker has in common with another moniker
12	RelativePathTo	Constructs a relative moniker between this moniker and another
13	GetDisplayName	Returns the display name
14	ParseDisplayName	Converts a display name to a moniker
15	IsSystemMoniker	Checks whether the moniker is one of the system-supplied types

The **IPersistStream** interface has these methods.

	Method	Description
16	GetSizeMax	Retrieves the size of the stream needed to save the object.
17	IsDirty	Determines whether an object has changed since it was last saved to its stream.
18	Load	Initializes an object from the stream where it was saved previously.
19	Save	Saves an object to the specified stream.

The **IPersist** interface has these methods.

	Method	Description
20	GetClassID	Retrieves the class identifier (CLSID) of the object.

Monikers are developed for getting custom activation objects. As we know CoCreateInstance API method internally calls IClassFactory methods to activate and use a COM component. Thus CoCreateInstance is useful only for those components which support IClassFactory interface. Recall that CoCreateInstance API method internally calls CoGetClassObject and DllGetClassObject API methods of the COM component. If everything goes fine then an instance of the IClassFactory is returned to the CoCreateInstance API. Finally, CoCreateInstance API

calls CreateInstance method on the IClassFactory instance to create the CoClass instance (by calling its empty constructor) and fetch the desired interface pointer.

Here, the problem is neither CoCreateInstance nor CreateInstance accepts an extra parameter that can be passed to the CoClass's constructor. So the restriction on server developer is to write a non-parameterized constructor only.
This is fine if no arguments are required in the constructor, but consider a situation - Device drivers are written in C/C++ using inline-assembly. These drivers start at boot time with the kernel and sound, video, camera apps call these device drivers by sending software interrupts. With unions as parameters, this strategy is unique for every operating system. Now to incorporate above strategy into COM to initialize the device driver, component constructor must accept arguments. Now here comes the moniker terminology. Moniker does not use IClassFactory internally and instead it encourages server developers to develop their own class factories. This way developers can design components such way that their constructor accepts arguments.

Windows WDM (Windows Driver Model) is built using moniker technology.

Also, Windows multimedia programming such as audio/video capture, audio/video encoding, and audio/video multi-format converters highly uses Moniker technology. These features use Microsoft's DirectX technology which extensively use moniker technique.

When VB uses a COM component, it calls CreateObject API method which internally calls CoCreateInstance API. Behind the screen, CoCreateInstance calls GetObject API method of VB which is nothing but moniker.

Three important API methods related to monikers:

```
HRESULT MkParseDisplayName(
  _In_  LPBC      pbc,

  // A pointer to the IBindCtx interface on the bind context object to be used in this
  // binding operation.

  _In_  LPCOLESTR szUserName,

   // A pointer to the display name to be parsed.

  _Out_ ULONG     *pchEaten,

   // A pointer to the number of characters of szUserName that were consumed. If the
   // function is successful, *pchEaten is the length of szUserName; otherwise, it is the
   // number of characters successfully parsed.

   Out_ LPMONIKER *ppmk

   // The address of the IMoniker* pointer variable that receives the interface pointer to
   // the moniker that was built from szUserName. When successful, the function has
   // called AddRef on the moniker and the caller is responsible for calling Release. If an
   // error occurs, the specified interface pointer will contain as much of the moniker
   // that the method was able to create before the error occurred.
);

HRESULT BindToObject(
  [in]  IBindCtx *pbc,

   // A pointer to the IBindCtx interface on the bind context object, which is used in this
   // binding operation. The bind context caches objects bound during the binding process,
   // contains parameters that apply to all operations using the bind context, and provides
   // the means by which the moniker implementation should retrieve information about its
   // environment.

  [in]  IMoniker *pmkToLeft,

   // If the moniker is part of a composite moniker, pointer to the moniker to the left of
   // this moniker. This parameter is primarily used by moniker implementers to enable
   // cooperation between the various components of a composite moniker. Moniker clients
   // should use NULL.
```

```
  [in]  REFIID   riidResult,

   // The IID of the interface the client wishes to use to communicate with the object that
   // the moniker identifies.

    [out] void     **ppvResult

    // The address of pointer variable that receives the interface pointer requested
    // in riid. Upon successful return, *ppvResultcontains the requested interface pointer
    // to the object the moniker identifies. When successful, the implementation must
    // call AddRef on the moniker. It is the caller's responsibility to call Release. If an
    // error occurs, *ppvResult should be NULL.
);

HRESULT CreateBindCtx(
  _In_  DWORD reserved,

   // This parameter is reserved and must be

  0. _Out_ LPBC *ppbc

   // Address of an IBindCtx* pointer variable that receives the interface pointer to the
   // new bind context object. When the function is successful, the caller is responsible
   // for calling Release on the bind context. A NULL value for the bind context indicates
   // that an error occurred.
);
```

Sample Moniker program:

Applications structure		
1 MonikerDllServer (DLL)	Moniker DLL server with custom class factory	1. MonikerDllServer.h (header) 2. MonikerDllServer.cpp (source) 3. MonikerDllServer.def (exported functions)
2 ClientOfMonikerDllServer (self-executable)	Win32 client for moniker DLL server	1. MonikerDllServer.h (header) 2. ClientOfMonikerDllServer.cpp (source)

11.1 MonikerDllServer

MonikerDllServer.def

```
LIBRARY MonikerDllServer
EXPORTS
      DllGetClassObject   PRIVATE
      DllCanUnloadNow     PRIVATE
      DllRegisterServer   PRIVATE
      DllUnregisterServer PRIVATE
```

MonikerDllServer.h

```
#pragma once

class IOddNumber :public IUnknown
{
public:
   // IOddNumber specific method declarations
   virtual HRESULT __stdcall GetNextOddNumber(int *) = 0;// pure virtual
};
```

```
class IOddNumberFactory :public IUnknown
{
public:
   // IOddNumberFactory specific method declarations
   virtual HRESULT __stdcall SetFirstOddNumber(int, IOddNumber **) = 0;// pure virtual
};

// CLSID of OddNumber Component {0A37F4CE-E65D-41d7-9B6C-
68CCAB226C6B} const CLSID CLSID_OddNumber =
{0xa37f4ce,0xe65d,0x41d7,0x9b,0x6c,0x68,0xcc,0xab,0x22,0x6c,0x6b};

// IID of IOddNumber Interface
const IID IID_IOddNumber =
{0x831612d,0x7b54,0x4bc1,0xb0,0x6e,0xc6,0x33,0x65,0xac,0xdd,0xcc};

// IID of IOddNumberFactory Interface const IID
IID_IOddNumberFactory =
{0xbe04438b,0xb142,0x4c52,0x99,0x6b,0xec,0x6,0x84,0x78,0x16,0x81};
```

MonikerDllServer.cpp

```
#include<windows.h>
#include"MonikerDllServer.h"

// class declarations
class COddNumber :public IOddNumber
{
private:
   long m_cRef;
   int m_iFirstOddNumber;
public:
   // constructor method
   declarations COddNumber(int);
   // destructor method
   declarations ~COddNumber(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // this class's custom method
   BOOL _stdcall IsOdd(int);

   // IOddNumber specific method declarations (inherited)
   HRESULT __stdcall GetNextOddNumber(int *);
};

class COddNumberFactory :public IOddNumberFactory
{
private:
   long m_cRef;
public:
   // constructor method declarations
   COddNumberFactory(void);
   // destructor method declarations
   ~COddNumberFactory(void);

   // IUnknown specific method declarations (inherited)
   HRESULT __stdcall QueryInterface(REFIID, void **);
   ULONG __stdcall AddRef(void);
   ULONG __stdcall Release(void);

   // IOddNumberFactory specific method declarations (inherited)
   HRESULT __stdcall SetFirstOddNumber(int, IOddNumber **);
};
```

```
// global variable declarations
long glNumberOfActiveComponents = 0;// number of active components
long glNumberOfServerLocks = 0;// number of locks on this dll
HMODULE ghModule = NULL; // DLL instance handle

// DllMain
BOOL WINAPI DllMain(HINSTANCE hDll, DWORD dwReason, LPVOID Reserved)
{
   switch (dwReason)
   {
      case DLL_PROCESS_ATTACH:
         ghModule = hDll;
         break;
      case DLL_PROCESS_DETACH:
         break;
   }

   return(TRUE);
}

// Implementation Of COddNumber's Constructor
Method COddNumber::COddNumber(int iFirstOddNumber)
{
   // hardcoded initialization to anticipate possible failure of QueryInterface()
   m_cRef = 1;
   m_iFirstOddNumber = iFirstOddNumber;// initialize to user input

   InterlockedIncrement(&glNumberOfActiveComponents);// increment global counter
}

// Implementation Of COddNumber's Destructor
Method COddNumber::~COddNumber(void)
{
   InterlockedDecrement(&glNumberOfActiveComponents);// decrement global counter
}

// Implementation Of COddNumber's IUnknown's Methods
HRESULT COddNumber::QueryInterface(REFIID riid, void **ppv)
{
   if (riid == IID_IUnknown)
      *ppv = static_cast<IOddNumber *>(this);
   else if (riid == IID_IOddNumber)
      *ppv = static_cast<IOddNumber *>(this);
   else
   {
      *ppv = NULL;
      return(E_NOINTERFACE);
   }

   reinterpret_cast<IUnknown *>(*ppv)->AddRef();

   return(S_OK);
}

ULONG COddNumber::AddRef(void)
{
   InterlockedIncrement(&m_cRef);

   return(m_cRef);
}

ULONG COddNumber::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
```

```
    {
        delete(this);
        return(0);
    }

    return(m_cRef);
}

// implementation of custom method of this class
BOOL COddNumber::IsOdd(int iFirstOddNumber)
{
    if (iFirstOddNumber != 0 && iFirstOddNumber % 2 != 0)
        return(TRUE);
    else
        return(FALSE);
}

// Implementation Of IOddNumber's Methods
HRESULT COddNumber::GetNextOddNumber(int *pNextOddNumber)
{
    // variable declarations
    BOOL bResult;
    bResult = IsOdd(m_iFirstOddNumber);

    if (bResult == TRUE)
        *pNextOddNumber = m_iFirstOddNumber + 2;// this gives next odd number
    else
        return(S_FALSE);

    return(S_OK);
}

// Implementation Of COddNumberFactory's Constructor Method
COddNumberFactory::COddNumberFactory(void)
{
    m_cRef = 1;// hardcoded initialization to anticipate possible failure
of QueryInterface()
}

// Implementation Of COddNumberFactory's Destructor Method
COddNumberFactory::~COddNumberFactory(void)
{
    // code
}

// Implementation Of COddNumberFactory's IUnknown's Methods
HRESULT COddNumberFactory::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IOddNumberFactory *>(this);
    else if (riid == IID_IOddNumberFactory)
        *ppv = static_cast<IOddNumberFactory *>(this);
    else
    {
        *ppv = NULL;

        return(E_NOINTERFACE);
    }

    reinterpret_cast<IUnknown *>(*ppv)->AddRef();

    return(S_OK);
}
```

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

ULONG COddNumberFactory::Release(void)
{
   InterlockedDecrement(&m_cRef);

   if (m_cRef == 0)
   {
      delete(this);
      return(0);
   }
   return(m_cRef);
}

// Implementation Of COddNumberFactory's IOddNumberFactory's method
HRESULT COddNumberFactory::SetFirstOddNumber(int iFirstOddNumber, IOddNumber
**ppIOddNumber)
{
   // variable declarations
   HRESULT hr;
   COddNumber* pCOddNumber = new COddNumber(iFirstOddNumber);

   if (pCOddNumber == NULL)
      return(E_OUTOFMEMORY);

   hr = pCOddNumber->QueryInterface(IID_IOddNumber, (void**)ppIOddNumber);
   pCOddNumber->Release();

   return(hr);
}
// Implementation Of Exported Functions From This Dll
HRESULT __stdcall DllGetClassObject(REFCLSID rclsid, REFIID riid, void **ppv)
{
   // variable declaraions
   COddNumberFactory *pCOddNumberFactory = NULL;
   HRESULT hr;

   if (rclsid != CLSID_OddNumber)
      return(CLASS_E_CLASSNOTAVAILABLE);

   // create class factory
   pCOddNumberFactory = new COddNumberFactory;

   if (pCOddNumberFactory == NULL)
      return(E_OUTOFMEMORY);

   hr = pCOddNumberFactory->QueryInterface(riid, ppv); pCOddNumberFactory-
   >Release();// anticipate possible failure of QueryInterface()

   return(hr);
}

HRESULT __stdcall DllCanUnloadNow(void)
{
   if ((glNumberOfActiveComponents == 0) && (glNumberOfServerLocks == 0))
      return(S_OK);
   else
      return(S_FALSE);
}
```

```
STDAPI DllRegisterServer()
{
   HKEY  hCLSIDKey = NULL, hInProcSvrKey = NULL;
   LONG  lRet;
   TCHAR szModulePath[MAX_PATH];
   TCHAR szClassDescription[] = TEXT("Simple Moniker class");
   TCHAR szThreadingModel[] = TEXT("Apartment");

   __try
   {
      // Create a key under CLSID for our COM server.

            lRet = RegCreateKeyEx(HKEY_CLASSES_ROOT, TEXT("CLSID\\{0A37F4CE-E65D-
            41d7-9B6C-68CCAB226C6B}"), 0, NULL, REG_OPTION_NON_VOLATILE,
            KEY_SET_VALUE | KEY_CREATE_SUB_KEY, NULL, &hCLSIDKey, NULL);

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

      // The default value of the key is a human-readable description of the coclass.

            lRet = RegSetValueEx(hCLSIDKey, NULL, 0, REG_SZ, (const
            BYTE*)szClassDescription, sizeof(szClassDescription));

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

      // Create the InProcServer32 key, which holds info about our coclass.

            lRet = RegCreateKeyEx(hCLSIDKey, TEXT("InProcServer32"), 0, NULL,
            REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hInProcSvrKey, NULL);

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

      // The default value of the InProcServer32 key holds the full path to our DLL.

      GetModuleFileName(ghModule, szModulePath, MAX_PATH);

      lRet = RegSetValueEx(hInProcSvrKey, NULL, 0, REG_SZ, (const
         BYTE*)szModulePath, sizeof(TCHAR) * (lstrlen(szModulePath) + 1));

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);

      // The ThreadingModel value tells COM how it should handle threads in our DLL.

      lRet = RegSetValueEx(hInProcSvrKey, TEXT("ThreadingModel"), 0,
         REG_SZ, (const BYTE*)szThreadingModel, sizeof(szThreadingModel));

      if (ERROR_SUCCESS != lRet)
         return HRESULT_FROM_WIN32(lRet);
   }
   __finally
   {
      if (NULL != hCLSIDKey)
         RegCloseKey(hCLSIDKey);

      if (NULL != hInProcSvrKey)
         RegCloseKey(hInProcSvrKey);
   }

   return S_OK;
}
```

```
STDAPI DllUnregisterServer()
{
        RegDeleteKey(HKEY_CLASSES_ROOT, TEXT("CLSID\\{0A37F4CE-E65D-41d7-
        9B6C-68CCAB226C6B}\\InProcServer32"));
   RegDeleteKey(HKEY_CLASSES_ROOT, TEXT("CLSID\\{0A37F4CE-E65D-41d7-9B6C-68CCAB226C6B}"));

   return S_OK;
}
```

11.2 ClientOfMonikerDllServer

MonikerDllServer.h: Use MonikerDllServer server's header file.

ClientOfMonikerDllServer.cpp

```
#include<windows.h>
#include<stdio.h>
#include"MonikerDllServer.h"

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// global  variable  declarations  IOddNumber
*pIOddNumber    =    NULL;    IOddNumberFactory
*pIOddNumberFactory = NULL;

// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmdLine, int nCmdShow)
{
   // variable declarations
   WNDCLASSEX wndclass;
   HWND hwnd;
   MSG msg;
   TCHAR AppName[] =
   TEXT("ComClient"); HRESULT hr;

   // COM Initialization
   hr = CoInitialize(NULL);
   if (FAILED(hr))
   {
              MessageBox(NULL, TEXT("COM Library Can Not Be Initialized.\nProgram Will
              Now Exit."), TEXT("Program Error"), MB_OK);
      exit(0);
   }

   // WNDCLASSEX initialization wndclass.cbSize =
   sizeof(wndclass); wndclass.style = CS_HREDRAW | CS_VREDRAW;
   wndclass.cbClsExtra = 0; wndclass.cbWndExtra = 0;
   wndclass.lpfnWndProc = WndProc; wndclass.hIcon =
   LoadIcon(NULL, IDI_APPLICATION); wndclass.hCursor =
   LoadCursor(NULL, IDC_ARROW); wndclass.hbrBackground =
   (HBRUSH)GetStockObject(WHITE_BRUSH); wndclass.hInstance =
   hInstance;


   wndclass.lpszClassName = AppName;
   wndclass.lpszMenuName = NULL;
   wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

   // register window class
   RegisterClassEx(&wndclass);

   // create window
   hwnd = CreateWindow(AppName,
```

```
        TEXT("Client Of COM Dll Server"),
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL);

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // COM Un-initialization
    CoUninitialize();
    return((int)msg.wParam);
}

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    // function declarations
    void SafeInterfaceRelease(void);

    // variable declarations
    IBindCtx *pIBindCtx = NULL;
    IMoniker *pIMoniker = NULL;
    ULONG uEaten;
    HRESULT hr;
    LPOLESTR szCLSID = NULL;
    wchar_t wszCLSID[255], wszTemp[255], *ptr;
    int iFirstOddNumber, iNextOddNumber; TCHAR
    str[255];

    switch (iMsg)
    {
    case WM_CREATE:
        // create a BindContext
        if (hr = CreateBindCtx(0, &pIBindCtx) != S_OK)
        {
            MessageBox(hwnd, TEXT("Failed To Get IBindCtx Interface Pointer"), TEXT("Error"),
            MB_OK);
            DestroyWindow(hwnd);
        }

        // Get String From Of Binary CLSID
        StringFromCLSID(CLSID_OddNumber, &szCLSID);

        wcscpy_s(wszTemp, szCLSID);
        ptr = wcschr(wszTemp, '{');
        ptr = ptr + 1;// to remove first opening '{' from CLSID string
        wcscpy_s(wszTemp, ptr);

        // to remove last closing '}' from CLSID string
        wszTemp[(int)wcslen(wszTemp) - 1] = '\0';
                wsprintf(wszCLSID, TEXT("clsid:%s"), wszTemp);
```

```
// Get Moniker For This CLSID
hr = MkParseDisplayName(pIBindCtx, wszCLSID, &uEaten, &pIMoniker);
if (FAILED(hr))
{
   MessageBox(hwnd, TEXT("Failed To Get IMoniker Interface Pointer"), TEXT("Error"),
   MB_OK);
   pIBindCtx->Release();
   pIBindCtx = NULL;

   DestroyWindow(hwnd);
}

// Bind the moniker to the named object
hr = pIMoniker->BindToObject(pIBindCtx, NULL, IID_IOddNumberFactory,
(void**)&pIOddNumberFactory);

if (FAILED(hr))
{
   MessageBox(hwnd, TEXT("Failed To Get Custom Activation -
   IOddNumberFactory Interface Pointer"), TEXT("Error"), MB_OK); pIMoniker-
   >Release();
   pIMoniker = NULL;
   pIBindCtx->Release();
   pIBindCtx = NULL;

   DestroyWindow(hwnd);
}

// release moniker & Binder
pIMoniker->Release();
pIMoniker = NULL;
pIBindCtx->Release();
pIBindCtx = NULL;

// initialize arguments hardcoded
iFirstOddNumber = 57;
// call SetFirstOddNumber() of IOddNumberFactory to get the first odd number
hr = pIOddNumberFactory->SetFirstOddNumber(iFirstOddNumber, &pIOddNumber);

if (FAILED(hr))
{
   MessageBox(hwnd, TEXT("Can Not Obtain IOddNumber Interface"), TEXT("Error"),
   MB_OK);
   DestroyWindow(hwnd);
}

// Release IOddNumberFactory Interface
pIOddNumberFactory->Release();
pIOddNumberFactory = NULL;

// call GetNextOddNumber() Of IOddNumber To Get Next Odd Number Of First Odd Number
pIOddNumber->GetNextOddNumber(&iNextOddNumber);

// Release IOddNumber Interface
pIOddNumber->Release();
pIOddNumber = NULL;

// show the results
wsprintf(str, TEXT("The Next Odd Number From %2d Is %2d"), iFirstOddNumber,
iNextOddNumber);

MessageBox(hwnd, str, TEXT("Result"), MB_OK | MB_TOPMOST);

// exit the application
DestroyWindow(hwnd);
break;
```

```
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    }

    return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}

void SafeInterfaceRelease(void)
{
    if (pIOddNumber)
    {
        pIOddNumber->Release();
        pIOddNumber = NULL;
    }

    if (pIOddNumberFactory)
    {
        pIOddNumberFactory->Release();
        pIOddNumberFactory = NULL;
    }
}
```

This page intentionally left blank

12. COM Interop

.NET technology is a powerful way to build components and distributed systems for the enterprise it can also leverage existing COM components. Classic COM components interoperate with the .NET runtime through an *interop* layer that will handle all the plumbing between translating messages that pass back and forth between the managed runtime and the COM components operating in the unmanaged realm, and vice versa. Additionally, there are tools provided with the .NET framework that allow you to expose .NET components to COM aware clients as if they were plain-vanilla COM components. The COM Interop handles all the intricacies and plumbing under the covers.

12.1 (Runtime Callable Wrapper) Using Classic COM Components from .NET Applications

A .NET application that needs to talk to our COM component cannot directly consume the functionality that's exposed by it. So, we need to generate some metadata. This metadata layer is used by the runtime to glean out type information, so that it can use this type information at runtime to manufacture what is called as a Runtime Callable Wrapper (RCW). The RCW handles the actual activation of the COM object and handles the marshaling requirements when the .NET application interacts with it. The RCW also does tons of other chores like managing object identity, object lifetime, and interface caching. Object lifetime management is a very critical issue here because the .NET runtime moves objects around and garbage collects them. The RCW serves the purpose of giving the .NET application the notion that it is interacting with a managed .NET component and it gives the COM component in the unmanaged space, the impression that it's being called by a good old COM client. The RCW's creation & behavior varies depending on whether you are early binding or late binding to the COM object. Under the hood, the RCW is doing all the hard work and thunking down all the method invocations into corresponding v-table calls into the COM component that lives in the unmanaged world.

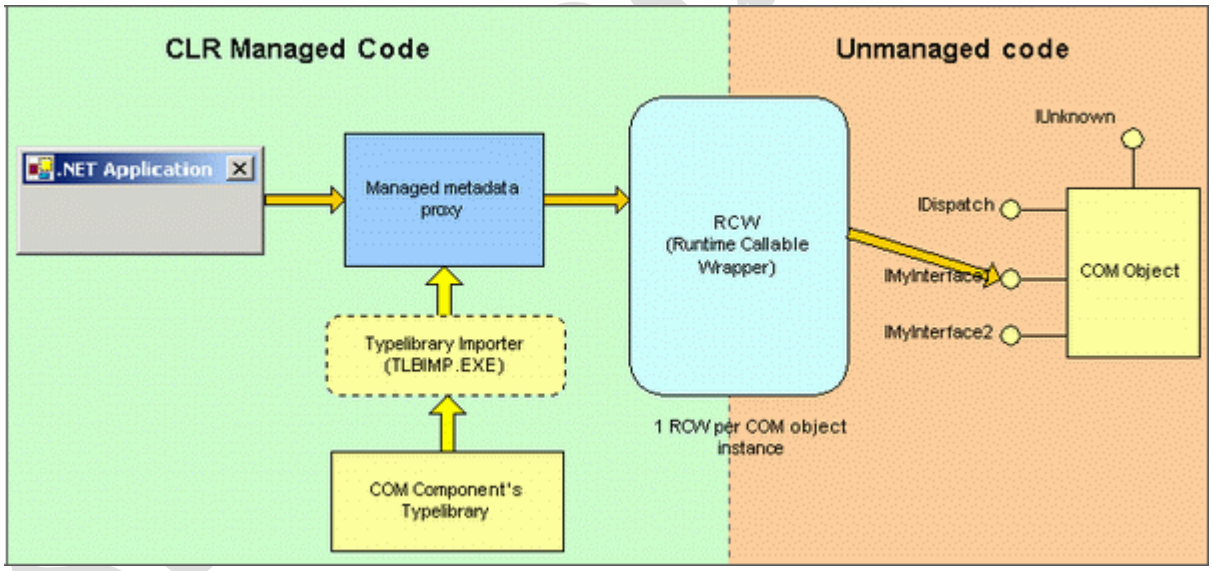


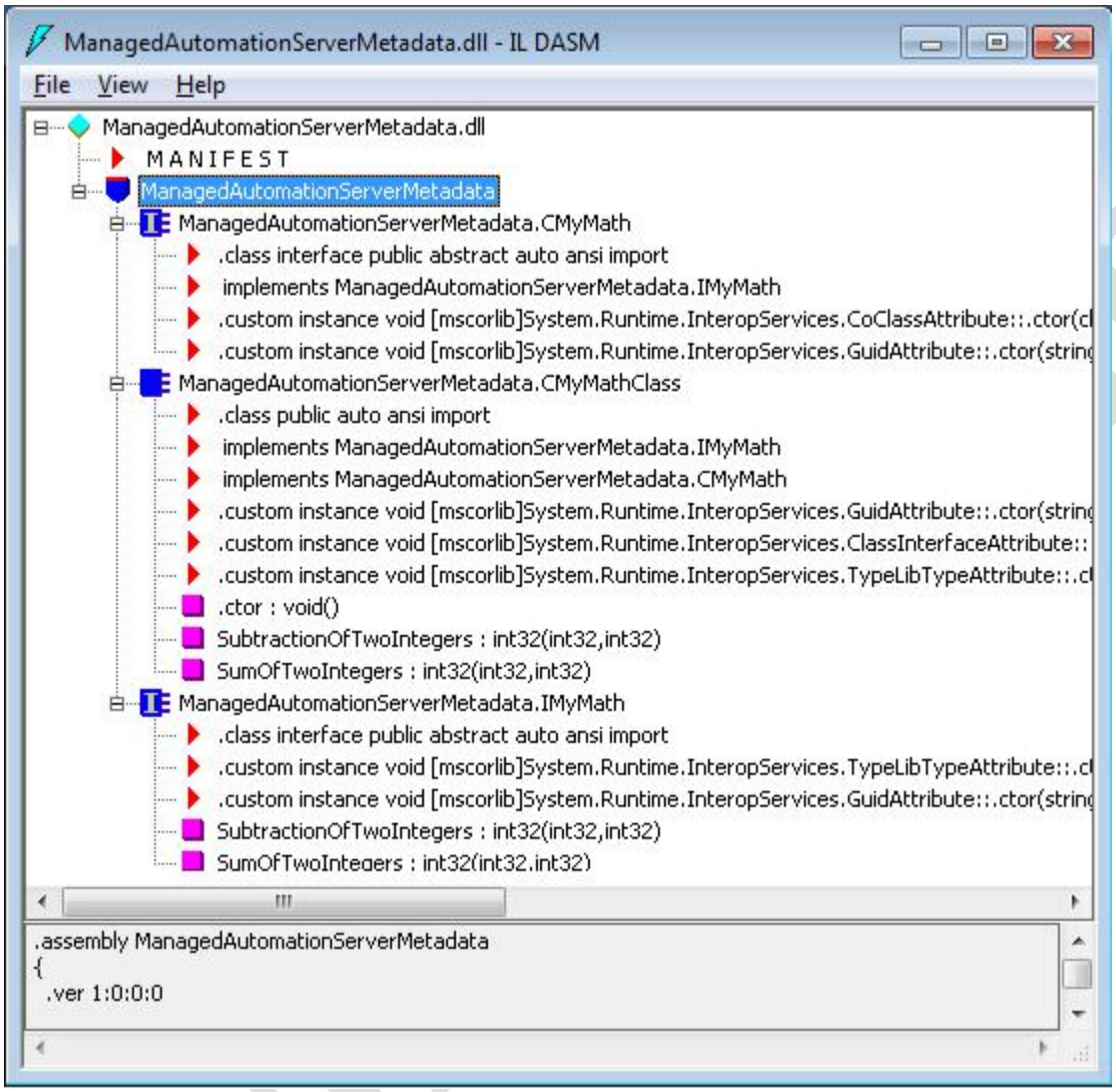
Figure 12.1: Consuming a Classic COM Component from a .NET application

The Type-library Importer utility (tlbimp.exe - ships with the .NET framework SDK) reads a type library and generates the corresponding metadata wrapper containing type information that the .NET runtime can comprehend.
From the command line, type the following command:

```
> TLBIMP AutomationServer.tlb /out:ManagedAutomationServerMetadata.dll
```

This command tells the type library importer to read your AutomationServer COM type library and generate a corresponding metadata wrapper called ManagedAutomationServerMetadata.dll. You can use ILDASM to view the metadata & the Intermediate language (IL) code generated for AutomationServer.tlb in ManagedAutomationServerMetadata.dll managed assembly. Go ahead and open ManagedAutomationServerMetadata.dll using ILDASM. Take a look at the metadata generated and you will see

that the SubtractionOfTwoIntegers and SumOfTwoIntegers methods are listed as a public method of the IMyMath interface that is implemented by the CMyMathClass class.



There is also a constructor (.ctor) that gets generated for the CMyMathClass class. The datatypes for the method parameters and return values have also been substituted to take their equivalent .NET counterparts. In our example, the SubtractionOfTwoIntegers and SumOfTwoIntegers methods' parameter with the *int* datatype has been replaced by the *int32* (an alias for System.Int32) datatype. Also notice that the parameter that was marked [out,retval] in the SubtractionOfTwoIntegers and SumOfTwoIntegers methods have been converted to the actual return value of the method (returned as a *int*). Any failure HRESULT values that are returned back from the COM object are thrown back as .NET exceptions.

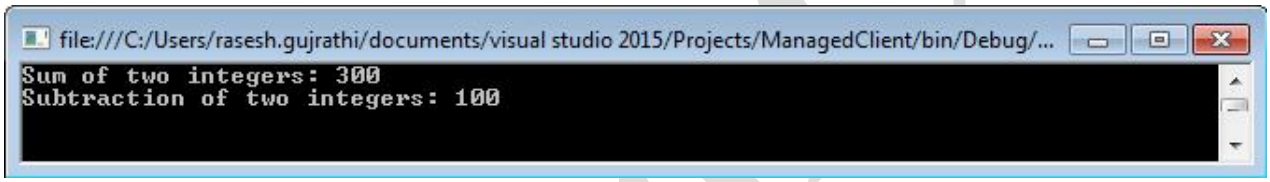
Following C# sample code shows how to use COM component and call its method using RCW

1. Add reference of the ManagedAutomationServerMetadata.dll in you C# project
2. Add following code in the C# source (.cs) file

```
using System;
using ManagedAutomationServerMetadata;
using System.Runtime.InteropServices;

namespace ManagedClient
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            try
            {
                IMyMath myMath = new CMyMathClass();
                int sum = myMath.SumOfTwoIntegers(100, 200);
                Console.WriteLine("Sum of two integers: " + sum);

                int subtraction = myMath.SubtractionOfTwoIntegers(200, 100);
                Console.WriteLine("Subtraction of two integers: " + subtraction);
            }
            catch (COMException comEx)
            {
                Console.WriteLine(comEx.Message);
            }
        }
    }
}
```

Here's how the output would look like:



Under the hood, the runtime fabricates an *RCW* and this maps the metadata proxy's class methods and fields to methods and properties exposed by the interface that the COM object implements. One RCW instance is created for each instance of the COM object. The .NET runtime only cares about managing the lifetime of the RCW and garbage collects the RCW. It's the RCW that takes care of maintaining reference counts on the COM object that it's mapped to, thereby, shielding the .NET runtime from managing the reference counts on the actual COM object. As shown in the ILDASM view, the CMyMathClass metadata class is defined under a namespace called ManagedAutomationServerMetadata. This class implements the IMyMath interface. All you need to do is, just create an instance of the CMyMathClass class using the *new* operator, and call the public class methods of the created object. When the method is invoked, the RCW thunks down the call to the corresponding COM method. The RCW also handles all the marshaling & object lifetime issues. To the .NET client, it looks nothing more than it's actually creating a managed object and calling one of its public class members. Anytime the COM method raises an error, the COM error is trapped by the RCW, and the error is converted into an equivalent *COMException* class (found in the *System.Runtime.InteropServices* namespace). Of course, the COM object still needs to implement the *ISupportErrorInfo* and *IErrorInfo* interfaces for this error propagation to work, so that the RCW knows that the object provides extended error information. The error is caught by the .NET client using the usual *try-catch* exception handling mechanism and has access to the actual error number, description, the source of the exception and other details that would have been available to any COM aware client. You could also return standard HRESULTs back and the RCW will take care of mapping them to the corresponding .NET exceptions to throw back to the client. For example, if you were to return a HRESULT of E_NOTIMPL from your COM method, then the RCW will map this to the .NET NotImplementedException exception and throw an exception of that type.

12.1.1 Dynamic Type Discovery

How does the classic QueryInterface scenario work from the perspective of the .NET client when it needs to check if the COM Object implements a specific interface? To QI for another interface, all you need to do is cast the object to the interface that you are querying for, and if it succeeds, your QI has succeeded as well. In case you attempt to cast the object to some arbitrary interface that the object does not support, a *System.InvalidCastException* exception is thrown, indicating that the QI has failed. It's that simple. Again, the RCW does all the hard work under the covers.

An alternate way to check if the object instance that you are currently holding supports or implements a specific interface type is to use C#'s 'is' operator. The 'is' operator does runtime type checking to see if the object can be cast safely to a specific type. If it returns *true*, then you can safely perform a cast to get the QI done for you. This way the RCW ensures that you are casting to only interfaces that are implemented by the COM object and

not just any arbitrary interface type. You can also use C#'s 'as' operator to cast from one type to another compatible type.

12.1.2 Late Bindings to COM Objects

All the examples that you saw above used the metadata proxy to early bind the .NET Client to the COM object. Though early binding provides a lot of benefits like strong type checking at compile time, providing auto-completion capabilities from type-information for development tools, and of course, better performance, there may be instances when you really need to late bind to a Classic COM object when you don't have the compile time metadata for the COM object that you are binding to. You can late bind to a COM object through a mechanism called *Reflection*. This does not apply to COM objects alone. Even .NET managed objects can be late bound and loaded using *Reflection*. Also, if your object implements a pure *dispinterface* only, then you are pretty much limited to only using Reflection to activate your object and invoke methods on its interface. For late binding to a COM object, you need to know the object's ProgID or CLSID. The CreateInstance static method of the System.Activator class allows you to specify the *Type* information for a specific class and it will automatically create an object of that specific type. But what we really have is a COM *ProgID* and COM *CLSID* for our COM object and not true .NET *Type* Information. So we need to get the *Type* information from the *ProgID* or *CLSID* using the GetTypeFromProgID or GetTypeFromCLSID static methods of the System.Type class. The System.Type class is one of the core enablers for Reflection. After creating an instance of the object using Activator.CreateInstance, you can invoke any of the methods/properties supported by the object using the System.Type.InvokeMember method of the Type object that you got back from Type.GetTypeFromProgID or Type.GetTypeFromCLSID. All you need to know is, the name of the method or property and the kind of parameters that the method call accepts. The parameters are bundled up into a generic System.Object array and passed away to the method. You would also need to set the appropriate *binding flags* depending on whether you are invoking a method or getting/setting the value of a property. That's all there is to late binding to a COM object.

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace ManagedClient
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                object objMathClassLateBound;
                Type objTypeMathClass;
                object[] arrayInputParams = { 100, 200 };

                // Here's how you use the COM PROGID to get the associated .NET
                // System.Type
                // objTypeMathClass =
                // Type.GetTypeFromProgID("ManagedServerForInterop.Math");

                // Here's how you use the COM CLSID to get the associated .NET
                // System.Type
                objTypeMathClass = Type.GetTypeFromCLSID(new Guid("{71169462-E879-
                4B9B-ACFF-FB718082D292}"));

                // create an instance of the object
                objMathClassLateBound = Activator.CreateInstance(objTypeMathClass);

                // Invoke SumOfTwoIntegers method
                int sum = (int)objTypeMathClass.InvokeMember("SumOfTwoIntegers",
                BindingFlags.Default | BindingFlags.InvokeMethod, null,
                objMathClassLateBound, arrayInputParams);
```

```
                System.Console.WriteLine("Sum of Two integers: " + sum);

                // Invoke SubtractionOfTwoIntegers method int subtraction
                = (int)
                objTypeMathClass.InvokeMember("SubtractionOfTwoIntegers",
                BindingFlags.Default | BindingFlags.InvokeMethod, null,
                objMathClassLateBound, arrayInputParams);

                System.Console.WriteLine("Subtraction of Two integers: " + subtraction);
            }
            catch (COMException comEx)
            {
                Console.WriteLine(comEx.Message);
            }
        }
    }
}
```

12.2 (COM Callable Wrapper) Consuming .NET Components from COM aware clients

In this section, we'll see how we can consume *managed .NET components* from *unmanaged COM aware clients*. The .NET framework allows fundamentally different or distinct applications in different platforms to talk to managed applications using wire protocols like *SOAP*. Unmanaged COM aware clients still get easier ways to talk to managed components. The .NET runtime allows *unmanaged COM aware clients* to seamlessly access .NET Components through the COM Interop and through the tools provided by the framework. This ensures that COM aware clients can talk to .NET components, as if they were talking to plain-vanilla Classic COM Components.

Too begin with, let's have a look at our simple Multiplication and Division implementation in .NET Component. Only classes that are public are added to the typelibrary and exposed to COM aware clients. Also, if the class needs to be creatable from a COM aware client, it needs to have a public default constructor. A public class that does not have a public default constructor still appears in the typelibrary, although it cannot be directly co-creatable from COM. The Math Component has two public methods MultiplicationOfTwoIntegers and DivisionOfTwoIntegers. It has public read-write properties called MultiplicationValue and DivisionValue defined with the corresponding get/set methods.

```
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace ManagedServerForInterop
{
    [ClassInterface(ClassInterfaceType.AutoDispatch)]
    public class Math
    {
        public int MultiplicationValue { get; set; }

        public int DivisionValue { get; set; }

        public Math()
        {
            // empty
        }

        public int MultiplicationOfTwoIntegers(int num1, int num2)
        {
            MultiplicationValue = num1 * num2;
            MessageBox.Show("Multiplication of 2 integers: " + MultiplicationValue);
            return MultiplicationValue;
        }

        public int DivisionOfTwoIntegers(int num1, int num2)
        {
            DivisionValue = num1 / num2;
```

```
            MessageBox.Show("Division of 2 integers: " + DivisionValue);
            return DivisionValue;
        }
    }
}
```

Notice that there is an attribute called *ClassInterface* that is applied to the *Math* class with its value set to *ClassInterfaceType.AutoDual*. For now, think of this as a way to tell typelibrary generation tools like REGASM.EXE and TLBEXP.EXE to export the public members of the .NET Component's class into a default Class Interface in the generated typelibrary. Using a Class Interface to expose the public methods of a .NET class is not generally recommended because of the versioning limitations. We'll take a look at how we can use *interfaces* explicitly to achieve the same thing. Defining an interface explicitly, deriving your .NET Component class from this interface and then implementing the interface's methods in your .NET Component is the recommended way of doing things if you are going to expose your .NET Component to COM aware clients.

Command line to build Math.dll:

```
$> csc /target:library /r:System.Windows.Forms.dll /out:Math.dll Math.cs
```

12.2.1 Generating a typelibrary from the assembly & Registering the assembly

We have generated a .NET assembly that other COM aware technologies cannot use directly. To make it happen, we need to extract type information from it so that other COM aware technologies can use it. To extract type information from it, you need to generate a typelibrary out of it. The .NET framework provides a couple of tools for this. You can use the Type Library Exporter utility (TLBEXP.exe) or the Assembly Registration Utility (Regasm.exe), both of which you'll find in the Bin directory of your .NET SDK installation. REGASM is a superset of the TLBEXP utility in that it also does much more than generating a typelibrary. It's also used to register the assembly, so that the appropriate registry entries are made to facilitate the COM runtime and the .NET runtime to hook up the COM aware client to the .NET component. You can use TLBEXP as well to generate the typelibrary, and then use REGASM to register the assembly.

```
$> regasm Math.dll /tlb:Math.tlb
```

The above call to REGASM.EXE makes the appropriate registry entries and also generates a typelibrary (Math.tlb) from the .NET assembly so that the typelibrary can be referenced from other client application.

12.2.2 Consuming the component from a C++ client

Let's have a plain C++ application that creates and invokes the .NET Component whose assembly we registered and generated a typelibrary from. Creation/activation of the component is the same as how you would create a COM object.

```
#import "Math.tlb" no_namespace,
raw_interfaces_only #include "Debug\math.tlh"

// #import creates two header files that reconstruct the type library contents in C++
// source code. The primary header file is similar to that produced by the Microsoft
// Interface Definition Language (MIDL) compiler, but with additional compiler-generated
// code and data. The primary header filehas the same base name as the type library, plus a
// .TLH extension. The secondary header file has the same base name as the type library,
// with a .TLI extension. It contains the implementations for compiler-generated member
// functions, and is included (#include) in the primary header file.

#include<windows.h>
#include <stdio.h> // for swprintf_s()

// global function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// WinMain
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmdLine, int nCmdShow)
{
```

```
    // variable declarations
    WNDCLASSEX wndclass;
    HWND hwnd;
    MSG msg;
    TCHAR AppName[] =
    TEXT("ComClient"); HRESULT hr;

    // COM Initialization
    hr = CoInitialize(NULL);

    if (FAILED(hr))
    {
       MessageBox(NULL, TEXT("COM Library Can Not Be Initialized.\nProgram Will Now Exit."),
       TEXT("Program Error"), MB_OK);
       exit(0);
    }

    // WNDCLASSEX initialization wndclass.cbSize =
    sizeof(wndclass); wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.cbClsExtra = 0; wndclass.cbWndExtra = 0;
    wndclass.lpfnWndProc = WndProc; wndclass.hIcon =
    LoadIcon(NULL, IDI_APPLICATION); wndclass.hCursor =
    LoadCursor(NULL, IDC_ARROW); wndclass.hbrBackground =
    (HBRUSH)GetStockObject(WHITE_BRUSH); wndclass.hInstance =
    hInstance;


    wndclass.lpszClassName = AppName;
    wndclass.lpszMenuName = NULL;
    wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // register window class
    RegisterClassEx(&wndclass);

    // create window
    hwnd = CreateWindow(AppName,
       TEXT("Client Of COM Dll Server"),
       WS_OVERLAPPEDWINDOW,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       NULL,
       NULL,
       hInstance,
       NULL);

    ShowWindow(hwnd, nCmdShow);

    UpdateWindow(hwnd);

    // message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }

    // COM Un-initialization
    CoUninitialize();

    return((int)msg.wParam);
}
```

```
// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
   // function declarations
   void ComErrorDescriptionString(HWND, HRESULT);

   // variable declarations
   HRESULT hr;
   int iNum1, iNum2, iMultiply;
   TCHAR str[255];

   switch (iMsg)
   {
   case WM_CREATE:
   {
      CLSID clsidMath;
      iNum1 = 10;
      iNum2 = 10;
      long iMultiply = 0;

      _Math *app;
      hr = CLSIDFromProgID(L"ManagedServerForInterop.Math", &clsidMath);

      if (FAILED(hr))
      {
         ComErrorDescriptionString(hwnd, hr);
         DestroyWindow(hwnd);
      }

      hr = CoCreateInstance(clsidMath, NULL, CLSCTX_INPROC_SERVER, __uuidof(_Math),
      (VOID**)&app);

      if (FAILED(hr))
      {
         ComErrorDescriptionString(hwnd, hr);
         DestroyWindow(hwnd);
      }

      hr = app->MultiplicationOfTwoIntegers(10, 10, &iMultiply);

      if (FAILED(hr))
      {
         ComErrorDescriptionString(hwnd, hr);
         DestroyWindow(hwnd);
      }

      wsprintf(str, TEXT("Multiplication Of %d And %d Is %d"), iNum1, iNum2, iMultiply);
      MessageBox(hwnd, str, TEXT("MultiplicationOfTwoIntegers"), MB_OK);

      /*_MathPtr pFoo(__uuidof(Math));

      hr = pFoo->MultiplicationOfTwoIntegers(10, 10, &iMultiply);

      if (FAILED(hr))
      {
         ComErrorDescriptionString(hwnd, hr);
         DestroyWindow(hwnd);
      }

      wsprintf(str, TEXT("Multiplication Of %d And %d Is %d"), iNum1, iNum2, iMultiply);
      MessageBox(hwnd, str, TEXT("MultiplicationOfTwoIntegers"), MB_OK); */

      DestroyWindow(hwnd);
   }
      break;
   case WM_DESTROY:
```

```
      PostQuitMessage(0);
      break;
   }

   return(DefWindowProc(hwnd, iMsg, wParam, lParam));
}

void ComErrorDescriptionString(HWND hwnd, HRESULT hr)
{
   // variable declarations
   TCHAR* szErrorMessage = NULL;
   TCHAR str[255];

   if (FACILITY_WINDOWS == HRESULT_FACILITY(hr))
      hr = HRESULT_CODE(hr);

   if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, hr,
   MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&szErrorMessage, 0, NULL) != 0) {

      swprintf_s(str, TEXT("%s"), szErrorMessage);
      LocalFree(szErrorMessage);
   }
   else
      swprintf_s(str, TEXT("[Could not find a description for error # %#x.]\n"), hr);

   MessageBox(hwnd, str, TEXT("COM Error"), MB_OK);
}
```

To enable the .NET Assembly Resolver to find the assembly housing your component, you will either need to place the component in the same directory as the application that's consuming it, or deploy the assembly as a Shared Assembly in the Global Assembly Cache (GAC). For now, just copy the Math.dll to the same directory as your C++ Client Application executable.

12.2.3 COM Interop

Let's take a peek at the registry entries that *Regasm.exe* made when we registered our assembly.

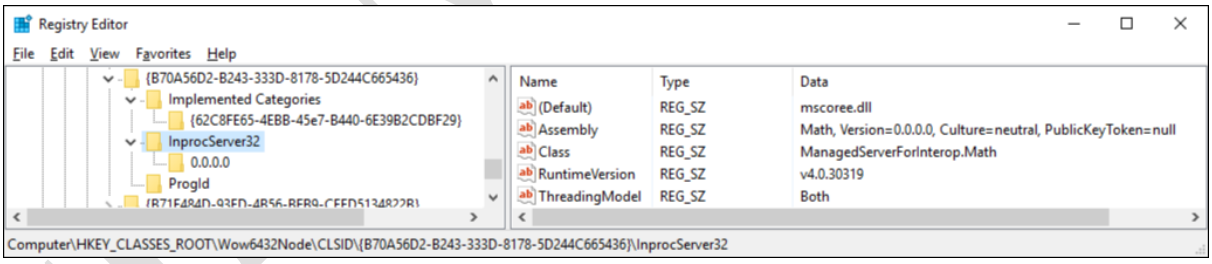


Figure 12.2: Registry Entries made during Assembly registration by REGASM

You can check your Component's CLSID in OLEVIEW.EXE by opening the typelibrary generated by REGASM. Check for the uuid attribute under the coclass section. If you navigate to your HKCR\CLSID\{...Component's CLSID...} key in the registry, you can see REGASM has made the relevant registry entries required by COM to activate an object hosted by an Inproc server. In addition, it has created a few other registry entries such as Class, Assembly, and RuntimeVersion that are used by the .NET runtime. The Inproc Server handler (indicated by the InProcServer32 key's default value) is set to mscoree.dll, which is the core CLR runtime execution engine. The COM runtime calls the DllGetClassObject entry point in MSCOREE.dll (The CLR runtime). The runtime then uses the Class ID (CLSID) passed to DllGetClassObject to look up the Assembly and Class keys under the InProcServer32 key to load and resolve the .NET assembly that will service this request. The runtime also dynamically creates a COM Callable Wrapper (CCW) proxy (a mirror image of the RCW) to handle the interaction between unmanaged code and the managed components. This makes COM aware clients think that they are interacting with Classic COM components and makes .NET Components think that they are receiving requests from a managed application. There is one CCW created per .NET component instance.

When a COM aware client interacts with a .NET component, the primary players are the CLR runtime and the COM Callable Wrapper (CCW) that gets fabricated by the .NET runtime. From then on, the CCW takes over and handles most of the spadework to get the two to work together. The CCW handles the lifetime management issues here. COM clients in the unmanaged realm maintain reference counts on the CCW proxy rather than on the actual .NET component. The CCW only holds a reference to the .NET component. The .NET Component lives by the rules of the CLR garbage collector as any other managed type would. The CCW lives in the unmanaged heap and is torn down when the COM aware clients no longer have any outstanding references to the objects. Just like the RCW, the CCW is also responsible for marshaling the method call parameters that move back and forth between the unmanaged client and managed .NET components. It's also responsible for synthesizing v-tables on demand. V-tables for specific interfaces are generated dynamically. They are built lazily only when the COM aware client actually requests a specific interface via a QueryInterfacecall. Calls on the CCW proxy are eventually routed away to a stub that actually makes the call into the managed object.

12.2.4 Generated TypeLibrary

Let's take a quick look at the kind of information that was put into the typelibrary generated by the Assembly Registration utility (REGASM). Open the typelibrary through OLEVIEW's Type Library Viewer so that you can take a look at the IDL file that was reverse engineered from the typelibrary.

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: Math.tlb

[
  uuid(CA86E018-57E1-36C7-A658-D57C763DD690),
  version(1.0),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, "Math, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null")

]
library Math
{
    // TLib : // TLib : mscorlib.dll : {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
    importlib("mscorlib.tlb");
    // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface _Math;

    [
      uuid(B70A56D2-B243-333D-8178-5D244C665436),
      version(1.0),
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "ManagedServerForInterop.Math")
    ]
    coclass Math {
        [default] interface _Math;
        interface _Object;
    };

    [
      odl,
      uuid(78AD3A69-B07C-33D2-9DE7-E68D44A5069C),
      hidden,
      dual,
      nonextensible,
      oleautomation,
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "ManagedServerForInterop.Math")

    ]
    interface _Math : IDispatch {
        [id(00000000), propget,
          custom(54FC8F55-38DE-4703-9C4E-250351302B1C, 1)]
        HRESULT ToString([out, retval] BSTR* pRetVal);
```

```
        [id(0x60020001)]
        HRESULT Equals(
                        [in] VARIANT obj,
                        [out, retval] VARIANT_BOOL* pRetVal);
        [id(0x60020002)]
        HRESULT GetHashCode([out, retval] long*
        pRetVal); [id(0x60020003)]
        HRESULT GetType([out, retval] _Type**
        pRetVal); [id(0x60020004), propget]
        HRESULT MultiplicationValue([out, retval] long*
        pRetVal); [id(0x60020004), propput]
        HRESULT MultiplicationValue([in] long
        pRetVal); [id(0x60020006), propget]
        HRESULT DivisionValue([out, retval] long*
        pRetVal); [id(0x60020006), propput]
        HRESULT DivisionValue([in] long pRetVal);
        [id(0x60020008)]
        HRESULT MultiplicationOfTwoIntegers(
                        [in] long num1,
                        [in] long num2,
                        [out, retval] long* pRetVal);
        [id(0x60020009)]
        HRESULT DivisionOfTwoIntegers(
                        [in] long num1,
                        [in] long num2,
                        [out, retval] long* pRetVal);
    };
};
```

If you take a look at the coclass section, it specifies the default interface as the Class Name prefixed by an _(underscore) character. This interface is called the *Class Interface* and its methods comprise all the non-static public methods, fields, and properties of the class. The Class Interface gets generated because you tagged your .NET class with the *ClassInterface* attribute. This attribute tells typelibrary generation tools such as RegAsm.exe and TlbExp.exe to generate a default interface known as the Class Interface and add all the public methods, fields, and properties of the class into it, so that it could be exposed to COM aware clients.

If you do not tag the ClassInterface attribute to a .NET Component's class, a default Class Interface is still generated. But in this case, it's an IDispatch based Class Interface that does not include any type information for the methods exposed nor their DISPIDs. This type of Class Interface is only available to late binding clients. This effect is the same as that of applying the ClassInterfaceType.AutoDispatch value to theClassInterface attribute. The advantage of the AutoDispatch option is that, since the DISPIDs are not cached and not available as a part of the type information, they don't break existing clients when a new version of the component is released since the DISPIDs are obtained at runtime by clients using something like IDispatch::GetIDsOfNames.

Only public methods are visible in the typelibrary and can be used by COM clients. The private members don't make it into the typelibrary and are hidden from COM Clients. The public properties and fields of the class are transformed into IDL propget/propput types. The MultiplicationValue and DivisionValue properties in our example have both set and get accessor defined and so, both the propset and propget are emitted for this property. There is also another interface called *_Object* that gets added to the coclass. The Class interface also contains 4 other methods.

ToString
Equals
    GetHashCode
    GetType

These methods are added to the default Class Interface because it implicitly inherits from the System.Object class. Each one of the methods and properties that's added to the interface gets a DISPID that is generated automatically. You can override this DISPID with a user defined one by using the DispId attribute. You'll notice that the ToString method has been assigned a DISPID of 0 to indicate that it is the default method in the class Interface. This means that if you leave out the method name, the ToString method will be invoked.

Let's examine the various ways in which we can facilitate the generation of the implicit Class Interface. We'll start by taking a look at the effect of applying the *ClassInterfaceType.AutoDual* value to the ClassInterface attribute.

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Math
{
        .
        .
        .
}
```

Notice that the value (*positional parameter* value) assigned to the ClassInterface attribute is *ClassInterfaceType.AutoDual*. This option tells typelibrary generation tools (like RegAsm.exe) to generate the Class Interface as a dual interface and export all the type information (for the methods, properties etc. and their corresponding Dispatch IDs) into the typelibrary. Imagine what would happen if you decided that you want to add another public method to the class. This mutates the Class Interface that gets generated and breaks the fundamental interface immutabilty law in COM because the v-table's structure changes now. Late Bound clients also have their share of woes when they try to consume the component. The Dispatch IDs (DISPIDs) get regenerated because of the addition of the new method and this breaks late bound clients too. As a general rule, using ClassInterfaceType.AutoDual is evil since it is totally agnostic about COM Versioning. Let's take a look at the next possible value that you can set for your ClassInterface attribute. Tagging yourClassInterface attribute with a value of ClassInterfaceType.AutoDispatch forces typelibrary generation tools to avoid generating type information in the typelibrary. So, if you had your *Math* class tagged with the ClassInterface attribute as shown below:

```
[ClassInterface(ClassInterfaceType.AutoDispatch)]
public class Math
{
   .
   .
   .
}
```

then, the corresponding typelibrary generated by RegAsm.exe would have an IDL structure such as this:

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: Math.tlb

[
  uuid(CA86E018-57E1-36C7-A658-D57C763DD690),
  version(1.0),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, "Math, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null")

]
library Math
{
    // TLib : // TLib : mscorlib.dll : {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
    importlib("mscorlib.tlb");
    // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface _Math;

    [
      uuid(B70A56D2-B243-333D-8178-5D244C665436),
      version(1.0),
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "ManagedServerForInterop.Math")
    ]
    coclass Math {
        [default] interface _Math;
        interface _Object;
    };

    [
      odl,
```

```
      uuid(2489666B-69D8-3B78-AE54-399389C09C52),
      hidden,
      dual,
      oleautomation,
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "ManagedServerForInterop.Math")

    ]
    interface _Math : IDispatch {
    };
};
```

You will notice that the default interface is an IDispatch interface and neither the DISPIDs nor the method type information is present in the Typelibrary. This leaves the COM aware client to consume .NET Components using only late-binding. Also, since the DISPID details are not stored as a part of the type information in the typelibrary, the clients obtain these DISPIDs on demand using something like IDispatch::GetIDsOfNames. This allows clients to use newer versions of the components without breaking existing code. Using *ClassInterfaceType.AutoDispatch* is much safer than using ClassInterfaceType.AutoDual because it does not break existing client code when newer versions of the component are released, though the former allows only late binding. The recommended way of modeling your .NET component to be exposed to COM aware clients is to do away with the Class Interface itself and instead, explicitly factor out the methods you are exposing into a separate interface, and have the .NET component implement that interface. Using a Class Interface is a quick and easy way to get your .NET component exposed to COM aware clients. But it's not the recommended way. Let's try rewriting our Math component by factoring out the methods explicitly into an interface and see how the typelibrary generation differs:

```
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace ManagedServerForInterop
{
    // Define the IMath interface
    public interface IMath
    {
        int MultiplicationValue { get; set; }

        int DivisionValue { get; set; }
        int MultiplicationOfTwoIntegers(int num1, int num2);
        int DivisionOfTwoIntegers(int num1, int num2);

    }/* end interface IMath */

    [ClassInterface(ClassInterfaceType.AutoDispatch)]
    public class Math: IMath
    {
        public int MultiplicationValue { get; set; }

        public int DivisionValue { get; set; }

        public Math()
        {
            // empty
        }

        public int MultiplicationOfTwoIntegers(int num1, int num2)
        {
            MultiplicationValue = num1 * num2;
            MessageBox.Show("Multiplication of 2 integers: " + MultiplicationValue);
            return MultiplicationValue;
        }

        public int DivisionOfTwoIntegers(int num1, int num2)
        {
            DivisionValue = num1 / num2;
            MessageBox.Show("Division of 2 integers: " + DivisionValue);
            return DivisionValue;
```

```
        }
    }
}
```

Here's how the corresponding IDL file looks like for the generated typelibrary. Notice that the Math class's default interface is now the IMath interface that was implemented by the class.

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: Math.tlb

[
  uuid(CA86E018-57E1-36C7-A658-D57C763DD690),
  version(1.0),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, "Math, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null")

]
library Math
{
    // TLib : // TLib : mscorlib.dll : {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
    importlib("mscorlib.tlb");
    // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface IMath;
    interface _Math;

    [
      odl,
      uuid(DF79E2EF-425E-3B7E-8246-3904443D1C80),
      version(1.0),
      dual,
      oleautomation,
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "ManagedServerForInterop.IMath")

    ]
    interface IMath : IDispatch {
        [id(0x60020000), propget]
        HRESULT MultiplicationValue([out, retval] long*
        pRetVal); [id(0x60020000), propput]
        HRESULT MultiplicationValue([in] long
        pRetVal); [id(0x60020002), propget]
        HRESULT DivisionValue([out, retval] long*
        pRetVal); [id(0x60020002), propput]
        HRESULT DivisionValue([in] long pRetVal);
        [id(0x60020004)]
        HRESULT MultiplicationOfTwoIntegers(
                        [in] long num1,
                        [in] long num2,
                        [out, retval] long* pRetVal);
        [id(0x60020005)]
        HRESULT DivisionOfTwoIntegers(
                        [in] long num1,
                        [in] long num2,
                        [out, retval] long* pRetVal);
    };

    [
      uuid(B70A56D2-B243-333D-8178-5D244C665436),
      version(1.0),
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "ManagedServerForInterop.Math")
    ]
    coclass Math {
        [default] interface _Math;
```

```
        interface _Object;
        interface IMath;
    };

    [
      odl,
      uuid(2489666B-69D8-3B78-AE54-399389C09C52),
      hidden,
      dual,
      oleautomation,
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "ManagedServerForInterop.Math")

    ]
    interface _Math : IDispatch {
    };
};
```

This approach of factoring out the methods of the .NET class explicitly into an interface and having the class derive from the interface and implement it, is the recommended way for exposing your .NET Components to COM aware clients. The *ClassInterfaceType.None* option tells the typelibrary generation tools that you do not require a Class Interface. This ensures that the IMath interface is the default interface. Were you to not specify the ClassInterfaceType.None value for the Class Interface attribute, then the Class Interface would have been made the default interface. Here's the gist of what we learnt in this section:

Class Interfaces with ClassInterfaceType.AutoDual are COM version agnostic. Try to avoid using them.
Class Interfaces with ClassInterfaceType.AutoDispatch do not export type information and DISPIDs to the typelibrary. They are COM versioning friendly. They can be accessed from COM aware clients only through late-binding.
Class Interfaces are a hack. Try to avoid them if possible. Use explicit interfaces instead.
Use ClassInterfaceType.None to make your explicit interface the default interface.

Another interesting observation is the way REGASM and TLBEXP generate a mangled method name in the IDL by appending a '_' followed by a sequence number when you have overloaded methods in your .NET class or interface that you are exporting to a typelibrary.

Another interesting observation is the way REGASM and TLBEXP generate a mangled method name in the IDL by appending a '_' followed by a sequence number when you have overloaded methods in your .NET class or interface that you are exporting to a typelibrary. For example, if you had exposed the following interface in your .NET Component:

```
public interface MyInterface
{
    String HelloWorld();
    String HelloWorld(int nInput);
}
```

Then, the IDL corresponding to the typelibrary that REGASM/TLBEXP generated would look like this.

```
[
  .....
]
interface MyInterface : IDispatch {

    [id(0x60020000)]
    HRESULT HelloWorld([out, retval] BSTR* pRetVal);

    [id(0x60020001)]
    HRESULT HelloWorld_2([in] long nInput, [out,retval]
                         BSTR* pRetVal);
};
```

Notice the '_2' appended to the second HelloWorld method to distinguish it from the first one in the IDL file.

Page intentionally left blank

13. Reference Reading

Microsoft Developer Network (MSDN)

Inside COM by Dale Rogerson

Essential COM by Don Box

COM specifications