* Component Object Model (COM) is a binary-interface standard for software component 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, and Windows shell, DirectX and the Windows runtime.

**Overview**

* The essence of COM is a language-neutral way of implementing objects that can be used in environments different from the one in which they were created, even across machine boundaries.
* For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it forces component implementers to provide well-defined interfaces that are separated from the implementation.
* The different allocation semantics of languages are accommodated by making objects accountable for their own creation and destruction through reference-counting.
* Casting between different interfaces of an object is achieved through the QueryInterface method.
* The preferred method of inheritance within COM is the creation of sub-objects to which method calls are delegated.
* COM is a technology defined and implemented as standard only on Microsoft Windows and Apple’s Core Foundation 1.3 and later plug-in API, that in any case implement only a subset of the whole COM interface.
* For some applications, COM has been replaced at least to some extent by the Microsoft .NET framework, and support for Web services through the Windows Communication Foundation.
* However, COM objects can be used with all .NET languages through the .NET COM Interop.
* Networked DCOM uses binary proprietary formats, while WCF encourages the use of XML-based SOAP messaging.
* COM is very similar to other component interface technologies, such as CORBA and Java Beans, although each has its own strength and weaknesses.
* Unlike C++, COM provides a stable API that does not change between compiler releases.
* This makes COM interfaces attractive for object-oriented C++ libraries that are to be used by clients compiled using different compiler versions.

**History**

* One of the first methods of inter-process communication in Windows was Dynamic Data Exchange, first introduced in 1987, that allowed sending and receiving messages in so-called conversations between applications.
* Antony Williams, one of the most notable thinkers involved in the creation of the COM architecture, later 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.
* Object Linking and Embedding, Microsoft’s first object-based framework, was built on top of DDE and designed specifically for compound documents.
* It was introduced with Word 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 document: As changes are made to the spreadsheet within Excel, they appear automatically inside the Word document.
* In 1991, Microsoft introduced Visual Basic extensions with Visual Basic 1.0.
* A VBX is a packaged extension in the form of a dynamic-link library that allows objects to be graphically placed in a form and manipulated by properties and methods.
* These were later adapted to be used by other languages such as Visual C++.
* In 1992, when version 3.1 of Windows was released, Microsoft released OLE 2 with its underlying object model.
* The COM application binary interface was the same as the MAPI API, which was released in 1992.
* While OLE1 was focused on compound documents, COM and OLE 2 were designed to address software components in general.
* Text conversations and Windows messages have proved not to be flexible enough to allow sharing application features in a robust and extensible way, so COM was created as a new foundation and OLE changed to OLE 2.
* In 1994, OLE customs were introduced as the successor to VBX controls.
* At the same time, Microsoft stated that OLE 2 would just be known as OLE, and OLE was no longer an acronym but the name for all of the company’s technologies.
* In early 1996, OLE found a new use for OLE Custom Controls, extending their web browser’s capability to present content, renamed some parts of OLE relating to the Internet “ActiveX”, and gradually renamed all OLE technologies to ActiveX, except the compound document technology that was used in Microsoft Office.
* Later that year, DCOM was introduced as an answer to CORBA.

**Related Technologies**

* COM was the major development platform for Windows and, as such, influenced development of several supporting technologies, such as COM+ and DCOM.
* For Microsoft to provide developers with support for distributed transactions, resource pooling, disconnected applications, event publication and subscription, better memory and processor management, as well as to position Windows as an alternative to other enterprise level operating systems, Microsoft introduced a technology called Microsoft Transaction Server on Windows NT 4.
* With Windows 2000, that significant extension to COM was incorporated into the operating system and renamed COM+.
* At the same time, Microsoft de-emphasized DCOM as a separate entity.
* Components that used COM+ services were handled directly by the added layer of COM+, in particular by operating system support for interception.
* In the first release of MTS, interception was tacked on – installing an MTS component would modify the Windows Registry to call the MTS Software, and not the component directly.
* Windows 2000 also revised the Component services control panel application used to configure COM+ components.
* An advantage of COM+ was that it could be run in “component farms”.
* Instances of a component, if coded properly, could be pooled and reused with its initialising routine without unloading it from memory.
* Components could also be distributed.
* COM+ and Microsoft Visual Studio provided tools to make it easy to generate client-side proxies, so although DCOM was used to make the remote call, it was easy to do so for developers.
* COM+ also introduced a subscriber/publisher Event mechanism called COM+ events and provided a new way of leveraging MSMQ with components called Queued Components.
* COM+ events extend the COM+ programming model to support late-bound events or method calls between the publisher/subscriber and the event system.

**.NET**

* Microsoft .NET provides means both to provide technology and to interact with COM+.
* .NET provides wrappers to most of the commonly used COM controls.
* Microsoft .NET hides most details from component creation and therefore eases development.
* Despite this, COM remains viable technology, providing high-performance, powerful and low-bandwidth consumption remoting and unparalleled stability due to its nearly two decades of improvement.
* Microsoft uses COM for modern operating system components like the ribbon control provided in Windows 7 and 8, besides hundreds of other COM components in their standard installation.
* It is also ideal for script control of applications such as Office or Internet Explorer since it provides an interface for calling COM object methods from a script, rather than requiring knowing the API at compile time.
* .NET can leverage COM+ by the System.EnterpriseServices namespace, and several of the services that COM+ provides have been duplicated in recent releases of .NET.
* For example, the System.Transactions namespace in .NET provides the TransactionScope class, which provides transaction management without resorting to COM+.
* Similarly, queued components can be replaced by Windows Communication Foundation with an MSMQ transport.
* There is limited support for backward compatibility.
* A COM object may be used in .NET by implementing a Runtime Callable Wrapper.
* .NET objects that conform to certain interface restrictions may be used in COM objects by a COM callable wrapper.
* From both the COM and .NET sides, objects using the other technology appear as native objects.
* WCF eases a number of COM’s remote execution challenges.
* For instance, it allows objects to be transparently marshalled by value across process or machine boundaries more easily.

**Windows Runtime**

* Microsoft’s new Windows Runtime Programming and application model is essentially a COM-based API, although it relies on an enhanced COM.
* Because of its COM-like basis, Windows runtime allows relatively easy interfacing from multiple languages, just as COM does, but it is essentially an unmanaged native API.
* The API definitions are, however, included in “.winmd” files, which are encoded in ECMA 335 metadata format, the same format that .NET uses with a few modifications.
* This common metadata format allows for insignificantly less overhead than P/Invoke when WinRT is invoked from .NET applications, and its syntax is much simpler.

**Security**

* COM and ActiveX components are run as native code on the user’s machine, with no sandboxing.
* There are therefore few restrictions on what the code can do.
* The prior practice of embedding ActiveX components on web pages with Internet Explorer did therefore lead to problems with malware infections.
* Microsoft recognised the problem with ActiveX as far back as 1996 when Charles Fitzgerald said, “We never made the claim upfront that ActiveX is intrinsically secure.
* Recent versions of Internet Explorer prompt the user before installing ActiveX controls, enabling the user to disallow the installation of controls from sites that the user does not trust.
* The ActiveX controls are signed with digital signatures to guarantee their authenticity.
* It is also possible to disable ActiveX controls altogether, or to allow only a selected few.
* The transparent process for out-of-process COM servers still promotes software safety in terms of process isolation.
* This can be useful for decoupling subsystems of large application into separate processes.
* Process isolation limits state corruption in one process from negatively affecting the integrity of the other processes, since they only communicate through strictly defined interfaces.
* Thus, only the affected subsystem needs to be restarted in order to regain valid state.
* This is not the case for subsystems within the same process where a rogue pointer in one subsystem can randomly corrupt other subsystems.

**Technical Details**

* COM programmers build their software using COM-aware components.
* Different components are identified by class IDs, which are Globally Unique Identifiers.
* Each COM component exposes its functionality through one or more interfaces.
* The different interfaces supported by a component are distinguished from each other using interface IDs, which are GUIDs too.
* COM interfaces have binding in several languages, such as C, C++, Visual Basic, Delphi, Python and several of the scripting languages implemented on the Windows platform.
* All access to components is done through the methods of the interfaces.
* This allows techniques such as inter-process, or even inter-computer programming.

**Interfaces**

* All COM components implement the IUnknown interface, which exposes methods for reference counting and type conversion.
* A custom IUnknown interface consists of a pointer to a virtual method table that contains a list of pointers to the functions that implement the functions declared in the interface, in the same order that they are declared in the interface.
* The in-process invocation overhead is therefore comparable to virtual method calls in C++.
* In addition to custom interfaces, COM also supports dispatch interfaces inheriting from IDispatch.
* Dispatch interfaces support late binding for OLE automation.
* This allows dispatched interfaces to be natively accessed from a wider range of programming languages than custom interfaces.

**Classes**

* A COM class is a concrete implementation of one or more interfaces, and closely resembles classes in object-oriented programming languages.
* Classes are created based on their class ID or based on their programmatic identifier string.
* Like many object-oriented programming languages, COM provides a separation of interface from implementation.
* This distinction is especially strong in COM, where objects cannot be accessed directly, but only through its interfaces.
* COM also has support for multiple implementations of the same interface, so that clients at runtime can choose which implementation of an interface to instantiate.

**Interface Definition Language and Type Libraries**

* Type libraries contain metadata to represent COM types.
* These types are described using Microsoft Interface Definition Language (IDL).
* IDL files define object-oriented classes, interfaces, structures, enumerations and other user defined types in a language-independent manner.
* IDL is similar in appearance to C++ declarations with some additional keywords such as “interface” and “library” for defining interfaces and collections of classes.
* IDL also supports the use of bracketed attributes before declarations to provide additional information, such as interface GUIDs and the relationship between pointer parameters and length fields.
* IDL files are compiled by the MIDL compiler.
* For C/C++, the MIDL compiler generates a compiler-independent header file containing struct definitions to match the vtbls of the declared interfaces and a C file containing the declarations of the interface GUIDs.
* C++ source code for a proxy module can also be generated by the MIDL compiler.
* This proxy contains method stubs for converting COM calls to remote procedure calls to enable DCOM for out-of-process communication.
* IDL file can also be compiled by the MIDL compiler into a type library.
* TLB files contain binary metadata that can be processed by different language compilers and runtime environments to generate language-specific constructs to represent the COM types defined in the TLB.
* For C++, it will convert the TLB back to its IDL representation.

**COM as an Object Framework**

* Because COM is a runtime framework, types have to be individually identifiable and specifiable at runtime.
* To achieve this, Globally Unique Identifiers are used.
* Each COM type is designated its own GUID for identification at runtime.
* In order for COM types to be accessible at both compile time and runtime, COM uses type libraries.
* It is through the effective use of type libraries that COM achieves its capabilities as a dynamic framework for the interaction of objects.
* Consider the following COM class definition in an IDL: The above code fragment declares a COM class named SomeClass which implements an interface name ISomeInterface.
* This is conceptually equivalent to defining the following C++ class: where ISomeInterface is a pure virtual C++ class.
* The IDL files containing COM interfaces and classes are compiled into type libraries files, which can later be parsed by clients at runtime to determine which interfaces an object supports and invoke an object’s interface methods.
* In C++, COM objects are instantiated with a CoCreateInstance function that takes the class ID and interface ID as arguments.
* Instantiation of some class can be implemented as follows:
* In this example, the COM subsystem is used to obtain a pointer to an object that implements ISomeInterface interface, and coclass CLSID\_SomeClass’s particular implementation of this interface is required.