Technology overview:

Microsoft’s Component Object Model (COM)
Summary

A component is a reusable piece of software in binary form (as opposed to source code) that can be plugged into other components from other vendors with relatively little effort. Microsoft has come up with COM, a specification of how to build components that can be dynamically interchanged, a standard that components and clients follow to ensure that they can operate together.

COM is very efficient, yet extremely flexible. It provides mechanisms for creating and controlling objects from scripting languages. It makes it possible for applications from one vendor to activate itself inside the user interface of another vendor’s application. It facilitates communication between components in different address spaces and threading models, and it takes the pain out of distributing objects across networks.

COM is a binary standard, and as such, it is not limited to only one programming language. COM components can be implemented in several different languages, including (but not limited to) C/C++, Visual Basic, Delphi and Java.

COM conforms well to the object-oriented paradigm. Also, COM enforces use of the two principles of reusable object-oriented design presented by the “Gang of Four”: “Program to an interface, not to an implementation” and “Favor object composition over class inheritance”.

COM is in use on 200 million systems worldwide. The market for COM components is expected to reach $3 billion in the year 2001. This base of available components allows developers to choose from a wide variety of components to assemble applications and solutions, which has revolutionized development on Windows platforms.
# Table of contents

SUMMARY ................................................................................................................................................. 2  
TABLE OF CONTENTS ............................................................................................................................... 3  
INTRODUCTION ............................................................................................................................................ 4  
  Why components? ......................................................................................................................................... 4  
  The evolution of COM ................................................................................................................................. 5  
    The clipboard ........................................................................................................................................... 5  
    DDE ........................................................................................................................................................... 5  
    16-bit OLE 1.0 .......................................................................................................................................... 5  
    Network DDE ......................................................................................................................................... 5  
    16-bit OLE 2.0 ......................................................................................................................................... 5  
    OLE controls ........................................................................................................................................... 5  
    32-bit OLE ............................................................................................................................................. 5  
    DCOM ..................................................................................................................................................... 5  
    ActiveX .................................................................................................................................................. 5  
THE MICROSOFT COM TECHNOLOGY ........................................................................................................... 6  
  What is COM? ................................................................................................................................................ 6  
  Interfaces .................................................................................................................................................... 7  
    Interface Definition Language .................................................................................................................. 7  
    IUnknown ............................................................................................................................................. 9  
    Custom interfaces .................................................................................................................................. 10  
    Dispatch interfaces ................................................................................................................................. 10  
    Dual interfaces ....................................................................................................................................... 10  
    Object creation ...................................................................................................................................... 12  
    Globally Unique Identifiers .................................................................................................................... 12  
    Self-registration ................................................................................................................................... 12  
    Class factories ...................................................................................................................................... 12  
  Component reuse ....................................................................................................................................... 13  
  Component reuse ....................................................................................................................................... 14  
    Containment ......................................................................................................................................... 14  
    Aggregation .......................................................................................................................................... 14  
  Server types .............................................................................................................................................. 16  
    In-process servers ................................................................................................................................. 16  
    Local servers ....................................................................................................................................... 16  
    Remote servers .................................................................................................................................... 17  
  Threading models ..................................................................................................................................... 18  
    Apartments ........................................................................................................................................... 18  
    ActiveX ............................................................................................................................................... 19  
    Automation .......................................................................................................................................... 19  
    Active Documents ............................................................................................................................... 19  
    ActiveX objects .................................................................................................................................... 19  
  Is COM object oriented? ............................................................................................................................ 20  
    Encapsulation ...................................................................................................................................... 20  
    Polymorphism ..................................................................................................................................... 20  
    Reuse ................................................................................................................................................ 20  
  OK, but what about…? ............................................................................................................................... 21  
    Cross-platform .................................................................................................................................... 21  
    Microsoft Transaction Server .............................................................................................................. 21  
    Microsoft Message Queue .................................................................................................................... 21  
    COM+ .................................................................................................................................................. 21  
    CORBA ............................................................................................................................................... 21  
CONCLUSION ............................................................................................................................................... 23  
REFERENCES .............................................................................................................................................. 24  
EXERCISES ................................................................................................................................................ 25
**Introduction**

**Why components?**

In the old days, an application usually consisted of a single monolithic binary file. It was not unusual that most, if not all, parts of the application were somehow aware of, and dependent on, other parts. A small modification in one part would mean that the entire system had to be re-tested again, to ensure that the modification did not have side effects somewhere else in the system.

Once the application had been compiled, it didn’t change until the next version was recompiled and shipped. With the current pace of change in the software industry, applications cannot afford to be static after they have been shipped. Developers had to find a way to breathe new life into applications that had already shipped. The solution was to break the monolithic application into separate pieces, or components. As technology evolves, new components can replace the existing out-dated components that make up the application.

Applications can be assembled very quickly by combining different components, very much like putting together bricks of LEGO. There will always be a need for some custom code that has to be developed in-house, but several pieces of most applications can often be bought as components from third party vendors.

The Giga Information Group estimated the market for COM components at $400 million a year in 1998 and they expect it to reach $3 billion in 2001. These market estimates are for third-party COM objects: they exclude COM components provided by Microsoft. This base of available components allows developers to choose from a wide variety of components to assemble applications and solutions, which has revolutionized development on Windows platforms.
The evolution of COM

The clipboard

The clipboard was the first and most fundamental method of sharing data between applications in Windows. A data-providing application described the data on the clipboard and a consuming application fetched it and used it.

DDE

The next step was *Dynamic Data Exchange* (DDE). It was a protocol for two applications to link pieces of data together dynamically, so that when one item of data in the source application changes, the linked item in the destination application would also change. DDE was based on the Windows messaging system and proved to be very slow, hard to write code for and not very robust or flexible at all.

16-bit OLE 1.0

The original goal of *Object Linking and Embedding* (OLE) was to support a document-centric view of the world where one could, for example, edit a spreadsheet from one vendor from within a word processor from another vendor (a technology known as compound documents). OLE 1.0 used DDE for the inter-process communication and the clipboard for copying data and information about the data source. The major advantage of OLE 1.0 was that you could activate an object in a compound document and edit it in the original application (which popped up as a separate window).

Network DDE

A short while after OLE 1.0 was released *Network DDE* emerged (in Windows NT 3.1). That was essentially DDE that allowed one side of the link to be on another machine across the network.

16-bit OLE 2.0

In 1993, OLE 1.0 became 16-bit OLE 2.0. It was based on the new *Component Object Model* (COM). OLE 2.0 was in fact the first COM system ever developed. As such, it is not an example of the best COM can do. OLE’s reputation of being big, slow and hard to code can only be blamed on the way OLE was implemented, not on the use of COM. The most exciting thing introduced by OLE 2.0 was in-place activation, which means that when an object is selected in a compound document, the object’s server (the application used for creating the object) is launched and run inside the document window (and not in its own window as was the case with OLE 1.0).

OLE controls

Visual Basic 3 could be extended with *Visual Basic Extension controls* (VBXs). Visual Basic 4 introduced *OLE Controls* (OCXs) as a more extensible architecture for components that could be used by other applications. An OCX is implemented as a DLL and it uses COM to communicate with its container application.

32-bit OLE

Windows NT 3.51 brought 32-bit OLE. The COM infrastructure was improved and layered completely on top of Microsoft’s Remote Procedure Call (RPC) system. The 2.0 and the term “Object Linking and Embedding” was dropped. The technology was from now on called OLE (pronounced ‘oley’) and the acronym did not longer have a meaning.

DCOM

With Windows NT 4.0 came *Distributed COM* (DCOM), which made COM calls across the network possible.

ActiveX

In 1995, Microsoft announced their strategy for the Internet, and the strategy included the new term *ActiveX*. ActiveX is a collection of additional higher-level technologies that are built on top of COM and its infrastructure.
The Microsoft COM technology

What is COM?

“COM is a specification and a set of services that allows you to create modular, object-oriented, customizable and upgradable, distributed applications using a number of programming languages.”

Let us take a moment to analyze the meaning of this statement:

“COM is a specification”

COM is not a programming language, nor an API. COM is a specification, which means that it is really a document describing what a component is, how a component can manage its own lifetime, and how it tells the outside world what it can do. COM provides a standard that components and their users (clients) follow to ensure that they can operate together.

“a set of services”

There is more to COM than just the specification. COM also provides an API, the COM Library, which provides component management services useful to both clients and components.

“modular, object-oriented, customizable and upgradable, distributed applications”

An application consisting of components that only interact through their interfaces is by definition modular. COM components are, as we shall discover later on, strictly object-oriented. Applications using components can be customized and upgraded by replacing specific components with others that support the same interfaces. COM also provides network transparency between clients and components, which makes it easier to develop distributed applications.

“a number of programming languages”

COM specifies a binary operability standard, and as such, it does not require the use of a specific language. Any language that has support for arrays of function pointers, and can call functions through these pointers, can be used to call or create COM components. There are even ways to write COM components using scripting languages like e.g. VBScript and JScript.

The remainder of this paper aims to give the reader a brief understanding of the COM architecture.
Interfaces

An interface is the set of operations that can be performed on an object. Those who interact with an object must do so only through the object’s interface. In addition, they should not know anything about how the object carries out its tasks (i.e. its implementation), only what its tasks are (specified by its interface).

“One should program to an interface, not to an implementation”. These wise words come from the book Design Principles by the “Gang of Four”. Anyone familiar with object-oriented design principles and the usefulness of design patterns know this fundamental rule.

COM enforces this design principle, because in COM interfaces are everything. In a client’s view, a component is just a set of interfaces. The only way to communicate with a component is through its interfaces, and this ensures encapsulation. Use of interfaces is also what allows clients to treat different components in the same manner (known as polymorphism). If two different components support the same interface, a client can treat them polymorphically.

A major problem with attempts to write component-based software in the past has been versioning of interfaces. When a client depends on some external code, and the interface to that code suddenly changes, the client will most likely fail to operate. What is most important to remember about interfaces in COM is that they never, ever, change. The ability for a single component to support multiple interfaces is the key feature that enables COM components to evolve over time. By adding more interfaces to a component, additional functionality can be added without endangering the binary compatibility to old clients that depend on the original interfaces. For example, if a component exposes an interface named IDoSomething, and the author of the component wants to add functionality to the component that would affect this interface, the traditional way of dealing with this problem would be to add a new interface called IDoSomething2. Now, all the new clients who know about the added functionality can use the IDoSomething2 interface, while all old clients keep using the IDoSomething interface, and keep on running even with newer versions of the component.

The illustration below depicts the graphical notation of a COM component (or object) exposing three different interfaces. Keep in mind that each interface can have several methods.

Interface Definition Language

An interface is really nothing more than a virtual function table, as illustrated in the picture below.
In C++, the definition of an interface ICool (interface names usually start with a capital I to distinguish them from class names) could be implemented as an abstract base class:

class ICool : public IUnknown
{
  public:
    virtual HRESULT ChillOut(int x, int y, int* retval) = 0;
};

One of the most important aspects of COM is its language independence. To be able to use the interface defined above from other programming languages, we would have to define the interface differently in every programming language we would like to use the interface in. This would lead to chaos, since a single interface could have multiple correct definitions. The solution to this problem is the Interface Definition Language (IDL).

The Open Software Foundation originally developed IDL for their Distributed Computing Environment’s Remote Procedure Call (RPC) package. IDL helps RPC programmers to ensure that both the client and the component sides of a project adhere to the same interface. IDL is not a programming language; it is a language used only to define interfaces. Microsoft has adopted IDL for use in defining COM interfaces. While COM interfaces use only a subset of IDL, they do require several nonstandard extensions that Microsoft added to support COM. As always, there is no standard that Microsoft doesn’t think they can improve…

The IDL file for a component implementing the ICool interface, could look something like this:

import “unknwn.idl”;
//interface ICool
{
  object,
  uuid(32bb8323-b41b-11cf-a6bb-0080c7b2d682),
  helpstring(“ICool interface”),
  pointer_default(unique)
}
interface ICool : IUnknown
{
  HRESULT ChillOut([in] int x, [in] int y, [out, retval] int* retval);
};

After describing your interfaces and components in IDL, you run them through the interface compiler (MIDL). The MIDL compiler takes the IDL description of your interfaces and generates several C-language source files that can be used when implementing, or using, the component from C/C++. This might seem odd, since COM is supposed to be language independent. Why can’t the MIDL compiler generate interface definitions for other languages? Well, instead of having to update the MIDL compiler for every new language that comes along, Microsoft chose a far more extensible approach. The solution was to have MIDL generate interface definitions in a single, universal format that all languages (including interpretative languages and macro programming environments) can understand. These are called type libraries and they are compiled, binary versions of the IDL files that can be accessed programatically. A type library file provides type information about all components, interfaces, methods, properties, arguments and structures described in an IDL file.

In addition to the C/C++ interface definitions and the type library file, MIDL also generates proxy and stub code for the interfaces in the IDL file. Proxy/stubs are necessary when an interface is going to be accessed from another address space or across a network. These will be covered in a later section.
**IUnknown**

The client always communicates with the component through an interface. The client uses an interface even when asking the component for other interfaces. The most basic interface in COM is called IUnknown and it is called that way because it defines an interface all others can talk to without having to know the exact workings of the object. The IUnknown interface contains three methods and it is the base interface for all other COM interfaces. This means that if you implement any other interface, you must implement the methods of the IUnknown interface as a subset. Since all interfaces inherit from IUnknown, all COM interfaces can be treated polymorphically as IUnknown interfaces. When a COM object is created, the creator gets a pointer to the object’s IUnknown interface. All further interaction with the component goes from there. The IUnknown interface consists of three methods:

- **QueryInterface()**
  
  Used for querying the object for other interfaces.

- **AddRef()**
  
  Increases the object’s internal reference counter.

- **Release()**
  
  Decreases the object’s internal reference counter (a component usually unloads itself from memory when there are no more clients using it).

Since the object has to keep track of how many references there are, out there somewhere, referring to it, it is important that all clients behave nicely and really call the AddRef() and Release() methods. If a client forgets to call AddRef() or calls Release() too many times, the object might think there are no other clients left and destroys itself prematurely. If a client calls AddRef() too many times or forgets to call Release(), the object will never destroy itself, resulting in a memory leak.

Only programmers of “low level” languages (like C/C++) are subject to this agony. All this is hidden for users of “high level” languages such as Visual Basic. To relieve the C++ clients of the huge responsibility and to avoid such programming mistakes, the smart pointer was invented. It is a template class that acts like an interface pointer and it takes care of calling the AddRef() and Release() methods when appropriate.
**Custom interfaces**

In C++, in order for a client to invoke a method in an interface it has to instantiate the object, get a pointer to the interface of interest, and then call the desired method through the vtable. Interfaces called through vtables are called custom interfaces, and this approach of invoking a method is known as *early binding*. This type of call is extremely efficient and has about the same overhead as calling a regular C/C++ function.

**Dispatch interfaces**

In real life, there are many situations where a client cannot bind early, or binding early may place too much of a limitation on an application. A good example is interpretative environments such as Visual Basic. In this case, when the client executable is compiled, all the COM components and interfaces that it would like to work with aren’t yet known. Therefore, we really can’t bind early. If Visual Basic could only use early binding, Microsoft would have to recompile and redistribute it every time someone wrote a new COM component that was to be used from Visual Basic.

So what do we do? Bind late, of course. Late binding means that the client application will find out at runtime what to do with the object. The COM implementers have specified a special interface, *IDispatch*, which can be used in late binding. Through this interface it is possible for the client to discover at runtime what it can do with the COM object. One disadvantage of binding late is, apart from its inefficiency, that it can cause run-time errors if a method that a client attempts to call doesn’t exist.

A macro language can’t get the offsets of the functions in an object’s vtable. All it knows is the name of the component, the name of the method, and the arguments to the method. This is when the *IDispatch* interface really comes in handy. It makes it possible for the macro language to call a method via the method’s name.

*Figure 4. Functions invocation via a dispatch interface*

The two most interesting methods of *IDispatch* are *GetIDsOfNames()* and *Invoke()*.

*GetIDsOfNames()* reads the name of a method and returns its *dispatch ID*. To execute the method, the client passes the dispatch ID to the *Invoke()* method. *Invoke()* then uses the dispatch ID as an index in an array of function pointers to call the method.

**Dual interfaces**

While the efficiency of early binding is very nice, so is also the flexibility of binding late. COM lets you make your object’s methods available to both early and late binders through *dual interfaces*. A dual interface is a dispatch interface that makes all the functions that are available through *Invoke()* also available directly through the vtable.
Dual interfaces allow C++ programs to make their calls via the vtable, such calls not only are easier for the C++ programmer to implement but execute faster. Macros and interpreted languages can also utilize the services of components implementing dual interfaces. Instead of calling through the vtable, they call via the Invoke method. A Visual Basic program can connect to either a dispatch interface or the vtable part of an interface. If you declare a variable in Visual Basic as type `object`, it connects through the dispatch interface. If you give the variable the type of the COM object, Visual Basic calls via the vtable.

Figure 5. A dual interface

<table>
<thead>
<tr>
<th>DISPID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“FunctionA”</td>
</tr>
<tr>
<td>2</td>
<td>“FunctionB”</td>
</tr>
<tr>
<td>3</td>
<td>“FunctionC”</td>
</tr>
</tbody>
</table>
Object creation

Creation (or instantiation) is very important in an object-oriented system. Objects must be created before they can be used. If object creation is different for every object, it becomes harder to use different objects polymorphically. Therefore, we want object creation to be as flexible as possible so that all components can be created in a similar manner. The more flexible the creation process is, the more easily it can be adapted to fit the needs of more components.

Globally Unique Identifiers

In COM, every component and interface is assigned a **globally unique identifier (GUID)**. A GUID is a 128-bit hexadecimal number that is guaranteed to be unique across space and time. A unique GUID is created by calling the Windows API function `CoCreateGuid()` or executing the `uuidgen.exe` program. These combine numerous pieces of information, including the time, date and network address to create a statistically unique 128-bit number. The concept of the GUID was originally developed by the Open Software Foundation for use in their Distributed Computing Environment (although they call them Universally Unique Identifiers or UUIDs).

In order for anyone else to use a component or an interface, it has to be made public. The GUIDs of a component and its interfaces are available to C/C++ developers in the header files generated by the MIDL compiler, and in the type library for everyone else. The GUID of a component is called a **class id (CLSID)** while the GUID of an interface is called an **interface id (IID)**.

To create an instance of a COM component, the client calls the COM Library function `CoCreateInstance()`, passing the CLSID as an argument to specify which component to create. The client passes an IID to the IUnknown interface method `QueryInterface()` to obtain a pointer to the desired interface.

Self-registration

Now, how does the COM Library find the component the client is asking for? All components are obligated to register information about themselves in the Windows registry. When a component is about to be instantiated, the registry is searched for a component with the specified CLSID. In the registry, the COM Library finds information about the component, like, for example, its type and where to find it in the file system.

Class factories

The `CoCreateInstance()` function does not itself create the instance of the component. Behind the scenes, it gets some help from a special kind of object called a **class factory**. Why do we need class factories? Why not simply call a magic function to directly instantiate the object? Class factories are necessary since object creation may require acquisition of system resources, coordination among objects, etc. The class factory is, in most cases, contained in the same DLL or EXE as the component. And since both the class factory and the component it creates are developed by the same person, the class factory can, and does have, special knowledge of the component it creates. With this approach, it’s only the factory object that has to be explicitly created by the COM subsystem.

Another responsibility of the object factory is to maintain location transparency. As we shall see later, COM components come in many flavors; in-process (same process), local-server (different address space) and remote-server (different machine). Clients must be able to reach objects easily, regardless of where they reside. If a component is running in a separate address space or on a remote computer, then the client can’t directly allocate memory for the component. The answer to this issue is the class factory.

The picture below illustrates what happens when a client creates an in-process component. The process of creating a local-server or a remote-server is, needless to say, more complicated.
Figure 6. Creation of an in-process object
Component reuse

COM does not support implementation inheritance because implementation inheritance binds one object closely to another object. If the implementation of a base object changes the derived objects break and must be changed too. The same “Gang of Four” as mentioned earlier also tell us to “favor object composition over class inheritance”. By not supporting implementation inheritance, COM enforces this design principle as well. COM developers have two flavors of object composition, containment and aggregation (or a mix thereof), to choose from for code reuse.

Containment

COM containment is similar to C++ containment. The outer component is simply a client of the inner components. The outer component implements its own interfaces using the inner components. The clients of the outer component do not know that the outer component uses other components to carry out its services.

The outer component can also re-implement an interface supported by an inner component by forwarding calls to the inner component. The outer component might specialize the interface by adding code before and after the code for the inner component.

Aggregation

Aggregation is a specialization of containment. When an outer component aggregates an interface of an inner component, it doesn’t re-implement the interface and explicitly forward calls to the inner component as is done with containment. Instead the outer component passes the inner component’s interface pointer directly to the client. The client then calls the interface belonging to the inner component directly.
The client must not know that it is talking to two different components, as that would break encapsulation. Making the outer and inner components behave as a single component is the trick to successful aggregation. Recall that a client obtains pointers to a component’s interfaces through the QueryInterface() method of the component’s IUnknown interface. Therefore, a component that supports being aggregated (being the inner component) has to be aware of when it is being aggregated. The trick is for the inner component to delegate the methods of its IUnknown interface to the outermost component (which is not always its aggregator). It is important to realize that aggregation can be nested. The reason why we always want to end up in the outermost component’s IUnknown is quite simple; it is the only object that is aware of the complete aggregation process and it alone knows what interfaces it is supposed to be exposing.
Server types

Another key feature of COM is that three different kind of objects are supported: in-process (DLL loaded into the client’s address space), local server (EXE in a separate address space on the same machine) and remote server (DLL or EXE on a different machine). Applications using COM components need not worry (or even know) what type of object it will end up using. You write the exact same code to hook up to in-process, local or remote objects. COM takes care of locating the component and starting processes or communicating over the network, if necessary.

In-process servers

An in-process server is implemented in a Dynamic Link Library (DLL) and is dynamically loaded into the client’s address space. In-process servers are fast and efficient and provide the highest performance of the three types of servers. The downside is that the component executes in the protected memory space of the client, with the same privileges as any other client code. This means that an errant object can crash the entire client process. The client has to be extremely trusting of the in-process server and very careful about how it calls it. If the client doesn’t want to trust the component, it has to call it through a surrogate object. A surrogate is a process that provides a wrapper for an in-process server so that it can be used as a local (out-of-process) server. COM supplies a standard surrogate that can be used to wrap in-process servers, but you could also write your own.

The picture below illustrates a client using an in-process component, loaded as a DLL into its own address space. The component’s class factory (supplied in the component’s DLL) is used to create an instance of the component. Also, the client has a pointer directly to the component’s interface.

![Diagram of in-process component](image)

Figure 9. In-process component loaded into a client’s address space

Local servers

Local servers are implemented as EXE files. When a client requests an object from a local server, the object that COM returns is implemented in another process. The advantage of this approach is that the memory of one process is protected from other processes. The processes implementing the client and server are separate, their memory is separate, yet the client can call the object’s methods, passing pointers to buffers in its own memory, and the server can access the data, and even change it. To the client, it is as if the object is in the address space of the client, just as it would be if it had been implemented as an in-process server.
How is this possible? Well, it’s COM (who else?) that does some magic stuff behind the scenes. On the server side, COM loads a stub DLL into the server’s address space. The stub can resolve the addresses in the server’s interface and call the object’s methods. On the client side, COM loads a proxy DLL into the client’s address space. The proxy provides code that looks just like the interface of the server, but it exists only to transmit the requests to call an interface’s methods to the stub in the server’s address space. Proxies are bits of code that take method calls, package them up along with any parameters, and send the packages via some inter-process or across-network communication method. On the other end, the stub unpackages the request, calls the object’s method, and then packages up any result to transmit back to the client. This packaging and unpackaging is called marshaling and unmarshaling. If the interface methods use only prescribed data types, the proxy and stub code can be generated automatically by the MIDL compiler. You can also write your own marshaling code if you want to optimize performance.

Remote servers

A local server is also capable of being accessed remotely. Without changing any code in the server (or the clients, for that matter) clients can instantiate it and call its methods across a network. It’s all a matter of how the system has been configured. Accessing COM objects remotely across a network is known as Distributed COM (DCOM). However, distributing objects across a network introduces tons of new, rather complicated, issues to worry about, such as security.
Threading models

Huge amounts of time can be spent describing threads and why many programmers fear them. This paper will not go into the details of threads and what implications they have on objects in terms of concurrency, reentrancy, etc. Instead, this section will give a brief overview of how COM provides a mechanism for clients and components that use different threading models to still work together.

Apartments

The basic unit of thread safety in COM is called an apartment. There are two types of apartments: the single-threaded apartment (STA) which can have only one thread executing, and the multi-threaded apartment (MTA) which can have more than one thread executing. A COM object belongs to exactly one apartment, and only threads within that apartment can execute methods on the object. This means that objects that live in a multi-threaded apartment have to be thread-safe (i.e. handle concurrent access, etc.). Objects that live in a single-threaded apartment don’t have to worry about concurrent access, since only one thread will ever enter the apartment.

An in-process component has information in the Windows registry about which types of apartments it is compatible with. When a client instantiates a component, COM ensures that the component is created and accessed in the correct type of apartment. If the client thread is in the wrong apartment type, COM will create the server object in a COM-managed thread of the correct apartment type. When this happens, the client will get a proxy to the server object that is appropriate for use in the client’s apartment. With this approach, even a non-thread-safe component can be accessed in a multi-threaded manner, since the proxy will serialize all calls to the actual server object. Note that this is only necessary for in-process components, since method calls to a local or remote component are always sent through a proxy anyway.

From the perspective of the client, all server objects appear to be using the client’s threading model, and from the perspective of the server, all clients appear to be using the server object’s threading model. This makes it possible to use components built with Visual Basic (which has limited threading options) from all clients, single or multi-threaded.
ActiveX

The ActiveX technology replaces what used to be known as OLE. ActiveX is a collection of additional higher-level technologies that build on COM and its infrastructure. The relationships and dependencies on the lower-level infrastructure of COM are shown in the picture below.

![Diagram of ActiveX, OLE, and COM relationships](image)

Figure 11. The relationship between ActiveX, OLE and COM.

The ActiveX technology has three important areas of functionality: Automation, Active Documents and ActiveX objects.

**Automation**

Automation (formerly known as OLE Automation) is the ability to create programmable applications that can be driven from an external script running in another application. Automation enables cross-application macro programming.

**Active Documents**

Active Documents provides the ability to embed information in a central document, encouraging a more document-centric user interface. Active Documents allow an application to open a document window from within another application. Internet Explorer is a good example of an application that supports applications that contain the Active Documents interfaces necessary to open documents within Internet Explorer windows (e.g. the Acrobat reader can open PDF documents inside an Internet Explorer window).

**ActiveX objects**

The ActiveX control specification defines the interfaces that must be implemented by a component to qualify as an ActiveX control that can be used in Visual Basic or other environments such as the Internet Explorer.
Is COM object oriented?

There have been many arguments about whether COM is object oriented or not. This section will show that COM indeed supports the three fundamental concepts of object orientation: encapsulation, polymorphism and reuse.

Encapsulation

COM hides all implementation details behind interfaces. The client sees only the server’s interface and knows nothing about the server’s internals. COM actually enforces a stricter encapsulation than many languages since a COM interface cannot expose public data members – all data access must happen through method calls.

Polymorphism

Simply put, polymorphism means that objects of different classes all look the same. COM’s usage of interfaces guarantees this. Objects that support the same interface are polymorphic in that interface. A client who can talk to e.g. an IDispatch interface can talk to any object that implements that interface (or interfaces derived from it).

Reuse

Reuse, you ask. Shouldn’t it be inheritance? No, it really should be reuse. Inheritance is not a fundamental OOP concept. Inheritance is merely how one expresses polymorphism in a programming language and how one achieves code reuse between classes. Inheritance is a means of polymorphism and reuse, not an end in itself. Polymorphism and reuse are the things one is really after.

COM supports, as described in an earlier section, code reuse by containment and aggregation. This form of reuse is often preferred over implementation inheritance, as it doesn’t break encapsulation. And, once again, as the “Gang of Four” so elegantly put it: “Favor object composition over class inheritance”.

OK, but what about…?
There is a lot more to COM than what can be covered within the scope of this paper. This section will glance at some of the neat stuff that hasn’t been covered.

Cross-platform
Most people claim that COM is for Windows use only. “If you need interaction between distributed objects in a heterogeneous environment, you have to go with CORBA”, they say. This is not true any more. While the CORBA people put an effort into getting CORBA out on as many platforms as possible from the beginning, Microsoft first concentrated on getting COM right on Windows, then to port it to other platforms. Today, COM has been ported to a number of Unix platforms, including Digital Unix and Linux.

Microsoft Transaction Server
Microsoft’s Transaction Server (MTS) provides an infrastructure for component based distributed applications. While DCOM allows objects to be distributed across a network, it doesn’t relieve the developer from the responsibility of taking care of issues such as security, threading and scalability. The goal of MTS is to provide the entire infrastructure for distributed applications so that developers only need to worry about their application-specific business problems. MTS provides support for transactions, eases threading concerns, simplifies security, makes installation easier (no need for component distribution since components execute “inside” MTS) and provides scalability through optimizations such as just-in-time (JIT) activation and instance pooling.

Microsoft Message Queue
Combining MTS and Microsoft’s Message Queue (MSMQ) means that not only will object interactions be performed under transactions, they are also guaranteed to be carried out. MSMQ will queue up requests and continue to call on an object until the method is called.

COM+
Windows 2000 introduced COM+, a technology that extends the Component Object Model to simplify the use of components. New features in COM+ include:

• Object pooling
  Enables a component to be configured in such a way so that instances of it are kept active in a pool, ready to be used by any client that requests the component. Significant performance and scalability enhancements can be achieved by reusing objects in this manner.

• Loosely coupled events
  Enables publishers of events to store event information in a public event store. Subscribers query this event store and select the events they are interested in. This has advantages over tightly coupled events where each subscriber has know which publisher to request notification from and what interfaces the publisher exposes. Loosely coupled events also has the advantage that it isn’t required that both the publisher and subscriber is running at all times.

• Queued components
  A mechanism based on MSMQ that provides an easy way to invoke and execute components asynchronously.

CORBA
As this paper concentrates on the fundamental low-level architecture of COM as a component technology, and not so much on its abilities as a middleware for distributed objects, I have deliberately
avoided making any comparisons whatsoever with CORBA. I think comparing COM and CORBA is like comparing apples and pears. They don’t address the same problem, as COM is mainly a specification of how to build components, while CORBA is a specification for a distributed object system. What could be an interesting comparison, though, would be DCOM versus CORBA, but that’s beyond the scope of this paper.
Conclusion

Today’s developers not only have to keep up with all the latest technologies and programming languages, they also have to build more and more complex systems and applications. Early applications, from the dawn of computing, were often simple and only needed to be good at one specific task. Modern applications often have to be “state of the art” in many aspects, including those outside the applications core functionality. For example, the developers in a company developing a system for supervising electrical power networks not only have to be experts at their core business (i.e. power networks), they also have to make their system able of presenting data in modern and ergonomically correct ways. This is (one situation) where third party components come in handy. Developers can concentrate on the core business logic of their applications and leave other things, e.g. presenting of 3D graphs, to those who are better at that.

Microsoft’s COM technology enables this approach of combining components into applications. It doesn’t matter who wrote the component or how it was implemented. Even components implemented in different programming languages can cooperate seamlessly with each other. As COM is being implemented on a continually growing number of platforms, DCOM is also becoming a serious candidate as a middleware in heterogeneous distributed systems.

COM is a constantly changing world. New technologies and tools are coming out faster than any one person can keep up with, let alone an entire industry. The pioneers of COM programming didn’t have as many fancy tools as we have today. Fortunately, Microsoft not only invents nicer and more easy-to-use tools for helping COM programmers (e.g. ATL, COM-savvy IDEs), they also keep on refining the technology itself (COM+).

The future of COM is bright!
References


Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, ISBN 0-201-63361-2
Exercises

1. Why is it important that an interface never changes, and how do you work around that problem if you need to modify an interface anyway?

2. COM does not support implementation inheritance. Name COM’s two strategies for code reuse.

3. Name the three different COM server types.