2

The Component Object Model

2.1 OVERVIEW

The goal of this chapter is to cover enough information on COM so that Eiffel developers can use the EiffelCOM wizard in an effective way. It will not cover all of COM since it would require an entire book but will present the main concepts needed to understand how to build an EiffelCOM system.

Briefly said, the Component Object Model is a Microsoft binary standard that establishes how two binary units can access each other at runtime. Such binary units can run in the same process, in different processes on the same machine or even on different machines. Components can be implemented in any language as long as the compiler produces COM standard compliant binaries.

The advantages of such an approach include an increased reusability (binary reuse), a better version management (COM standard implies that new component versions will still be compatible with older ones) and a built-in runtime environment. The binary reuse aspect of COM added with the source reuse ability of Eiffel offer the developer an ideal platform to increase considerably their productivity.

2.2 GENERALITIES

This paragraph will present briefly the principal notions of COM needed to understand the rest of this chapter. COM is a binary standard that describes how the component can communicate with the outer-world. Communication is done through well defined *interfaces*. Each interface is a *specification* of a group of methods and functions. An interface does not contain the implementation of these routines and functions but only their specification (signatures). The actual implementation lies in the *coclass*. There can be different implementations of a same interface in different coclasses. Finally each coclass can be instantiate using a *class object* or *class factory.* These three notions will be discussed further in the forthcoming paragraphs.

Interfaces

They are at the heart of any COM component. Interfaces are described in the definition file of a component. They consist of a group of semantically related

functions that can be accessed by the clients of the component. Although they are a specification they also have a physical representation. A client can request a pointer on an interface and will access the component functions through that pointer. Interfaces are the only possible way to access functions from a component. They enforce information hiding by providing only the public functions to the client.

Interfaces also define the *type* of a component. Each interface corresponds to a specific *view* of the component. It can be compared to polymorphism in the Object Oriented world. Whenever an interface from a component is requested, only the functions defined on that interface are accessible as if the component was polymorphically cast into an object of the type of that interface.

The COM specification requires that any interface provides access to all interfaces on the same component. All the interface should include a specific function called *QueryInterface* that will provide a pointer on any other interface of the component. Interfaces are identified with a globally unique identifier (GUID) guaranteed to be unique in time and space. Since this function has to be on every interface, it has been abstracted into a specific interface called *IUnknown* which all other interfaces must inherit from. The two other functions exposed by *IUnknown* are *AddRef* and *Release*. These functions should be called respectively when a client gets a reference on an interface or when it discards that reference. These two functions define the lifetime of the component: each interface keeps track of clients keeping a reference on them and when no clients have references anymore, the component *can* be unloaded from memory. You might start to worry thinking that this business of reference counting will imply lots of headaches, memory leaks, etc... and you would be right should you choose a low-level language to implement your components. Fortunately, you will never have to implement or use these functions in Eiffel: all the processing related to *IUnknown* is provided by the EiffelCOM runtime. Calls to *QueryInterface* are done “behind the scene” and only when needed. The lifetime of the component is also taken care of by the EiffelCOM runtime.

Coclass

We have seen that interfaces can be perceived as views of a component. This conceptual representation actually maps the implementation of an EiffelCOM component since the coclass inherits from the interfaces and implements their deferred features. Indeed, interfaces are deferred classes with all features accessible from outside deferred. The coclass is an Eiffel class that inherits from these interfaces and implements all the features. This design is not specific to Eiffel though and can be found in other languages as well. The coclass defines the behavior of the interfaces functions.

Class Object

We have seen that interfaces are accessed through interface pointers. But how does a client get hold on one of these?

The answer lies in the class object. The name of this module should really be coclass factory since its goal is to spawn instances of the coclass on request. Class objects are accessed by COM whenever a client request a new instance of the associated component. COM loads the class object and asks it to provide the interface pointer requested by the client.

The way a class object is loaded in memory (this process is called *activation*) depends on the location of the component (See Location for a description of the possible locations of a component). If the component is an in-process server then the class object is called directly through the functions exported from the DLL. If the component is an out-of-process server then it provides COM with a pointer to the class object. In both cases, once the component is loaded, COM has access to the class object and can call it would a client request a new instance of a component.

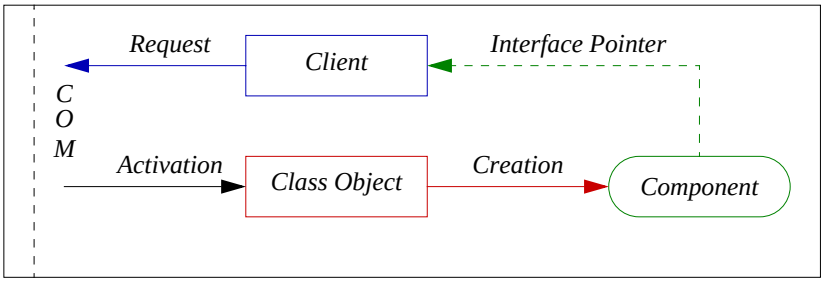

Figure 1: Component Creation

The code for the class object is generated by the EiffelCOM wizard so that Eiffel programmers will not have to worry about it.

2.3 TYPES OF COMPONENTS

ActiveX, DirectX, OCX, COM+, ADO+, ASP etc.... who never heard of these technologies? They all represent yet another use of the COM standard. This paragraph will focus on categorizing COM components according to their own properties as well as the context in which they are used. The categorization will define how the EiffelCOM wizard should be used to wrap or create a component.

Location

The first criteria that defines the type of component is from where it will be accessed: will the component be loaded in the client process or will the component be a remote server for a distributed application? In the former case, the component is compiled as a Dynamic Link Libraries (DLL) while in the latter case it is a standard executable.

In-process Components

Typical instances of DLL components are found in technologies such as OCX, ActiveX or ASP. These are small, downloadable binaries that will be loaded and

executed in a *container*. The container acts as a client of the component. The EiffelCOM wizard provides the ability to wrap such components by providing access to its interface to any Eiffel container. It is also possible to create a new In-process component in Eiffel. One main difference with out-of-process component (other than the nature of the module, DLL versus executable) is the way in-process components are activated. In the case of out-of-process components, the component will specify COM when it is ready to receive calls from client. In the case of an in-process server the call is coming directly from COM: COM first loads the DLL into the client process and then calls the exported function *DllGetClassObject* to access the component class object. The other three exported functions of an in-process component are *DllRegister* to register the component in the Windows registry, *DllUnregister* to unregister the component from the registry and finally *DllCanUnloadNow* which gets called by COM whenever it tries to unload the component from memory. These four functions must be accessible from outside the DLL for the in-process component to work properly.

Out-of-process Components

These components are standard executable acting as servers that can be accessed locally or over a network. Typically used in a three tier client server architecture, the major difference with in-process servers (other than the module type - executable instead of DLL) is their lifetime. In-process components are typically loaded to achieve a specific task and unloaded just after while out-of-process components are servers supposed to run continuously. The EiffelCOM wizard allows to build clients for such servers in Eiffel. It also provides the ability to create such servers.

Access Type

Regardless of its location, a COM components can be accessed either directly through its interfaces or by using Automation.

Automation

Automation consists in using a well known interface to provide access to a group of *methods* and *properties*. This interface called *IDispatch* includes the method *invoke* that allows to call a method, set or get a property on the Automation server. One advantage of that approach is that the interface is a *standard interface* whose functions and methods are specified. As a result, Windows can include a built-in marshaller for that interface (See Marshalling for information on what a marshaller is). The supported types (known as Automation types) and their Eiffel equivalents are listed in the following table:

COM Type	*Eiffel equivalent*	*Description*
boolean	*BOOLEAN*	Standard boolean
unsigned char	*CHARACTER*	Standard character

double	*DOUBLE*	Standard double
float	*REAL*	2 bytes real
int	*INTEGER*	Standard integer
long	*INTEGER*	Standard integer
short	*INTEGER*	2 bytes integer
BSTR	*STRING*	Standard string
CURRENCY	*ECOM_CURRENCY*	Currency value
DATE	*DATE*	Standard date
SCODE	*INTEGER*	Return status
Interface IDispatch *	*ECOM_QUERIABLE*	Automation interface
Interface IUnknown *	*ECOM_QUERIABLE*	Generic interface
dispinterface	*ECOM_QUERIABLE*	Automation interface
Coclass *Typename*	*TYPE_NAME*	Component main class
SAFEARRAY(*TypeName*)	*ARRAY [TypeName]*	Array
TypeName*	*CELL [TypeName]*	Pointer to type
Decimal	*ECOM_DECIMAL*	Decimal value

The other advantage is a more dynamic discovery of the methods and properties of a component at runtime. Indeed the *IDispatch* interface also includes methods to check whether a method or property is available and, in that case, get its identifier. This process is called *late binding* and allows component to discover at runtime what are other components functionality.

This approach has also a lot of drawbacks: firstly, *late binding* is not an efficient way of calling a function on an interface since its identifier must first be requested and then the function called. That's two round trips which can be expensive in a distributed environment. Secondly, since the marshaller is built-in, it has to know in advance all the possible types that a function can accept to be able to marshall the corresponding data. There are consequently a limitation on the number of types that one can use in signatures of functions on an Automation compatible interface. The set of available types is called *Variant* and cover most of the standard types. It does not allow however the passing of complex user defined data types. For these reasons Automation is mostly used in scripting environments (where speed is not an important factor) to accomplish simple tasks.

Direct Access

Direct interface access is the preferred way to access remote servers where speed becomes a concern and data types are specific to the application. The first interface

pointer on the component is obtained through the class object (see Class Object). Other interfaces on the component are obtained by calling the *QueryInterface* function.

As information on any interface cannot be accessed dynamically, the description of the interfaces must be provided to tools that need to handle the components such as the EiffelCOM wizard. The official way of describing components and interfaces is through IDL. Once an IDL file has been written to describe a component it can be compiled with *MIDL* to generate both a type library and the code for the marshaller specific to that interface.

EiffelCOM

The idea in EiffelCOM is that the way a component is accessed is implementation detail that the user should not have to deal with. Of course he should be able to choose what kind of access he wants to use but this choice should have no impact on the design of the Eiffel system itself. For that reason, the Eiffel code generated by the wizard follows the same architecture independently of the choice made for interface access and marshalling. The difference lies in the runtime where the actual calls to the components are implemented.

2.4 DEEPER INTO COM

The next paragraph gives a bit more details on the internals of COM. The understanding of these details are not required to use the EiffelCOM wizard but might help making better decisions when designing new EiffelCOM components.

Apartments

The first interesting subject that requires more in-depth cover is the execution context of a component. Components can be run in the same process as the client but can also run in a separate process even on a different machine.

This superficial description only take into accounts processes. What happens if a component uses multithreading to achieve it tasks? In a case of a remote server, this scenario does not seem too esoteric. The problem is that a server does not (and should not) know in advance what its clients will be. It cannot assume that the client will be able to take advantage of its multithreading capabilities. Conversely a multithreaded client should not rely on the server ability to handle concurrent access.

The solution chosen in the COM specification is to define an additional execution context called an *apartment*. When COM loads a component it creates the apartment in which the component will run. Multiple instances of a multithreaded component will leave together in the same apartment since asynchronous calls will be handled correctly and there is no need to add any synchronization layer. On the other hand, singlethreaded component will be alone in their apartment and any concurrent calls coming from clients will be first synchronized before entering the apartment. These

two behaviors define two different kinds of apartments: *Multi Threaded Apartments* (MTA) and *Single Threaded Apartments* (STA).

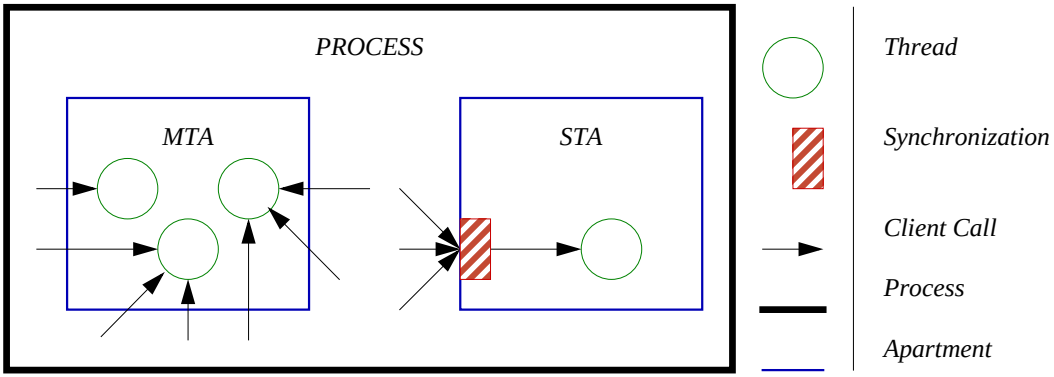

Figure 2: Apartments

Apartments solve the problem of concurrency by removing the necessity of knowing the multithreaded capability of a component and its clients. Multithreaded clients can always make asynchronous calls and depending on whether the component handles concurrent access or not, they will be forwarded or first synchronized. There can be multiple instances of STA running in one process while there will be at most one MTA.

Marshalling

At this point you might wonder how calls can “cross” the apartments boundaries. Components from a STA can make calls to components running in a MTA and vice versa. These apartments might be running in different processes or even on different machines. The approach chose in the COM specification is using the *proxy* and *stub* patterns.

The idea is to trick the client of an interface by providing an interface proxy in its apartment. The proxy include exactly the same function as the interface itself but their implementation will just forward the call to the actual interface. The client has no idea whether the entity it is dealing with is the actual interface or just a proxy. One of the main interest of that approach is that the client implementation is independent from the location of the component.

Last explanation is not totally accurate: the call will not be forwarded to the actual interface but to its stub. The stub is the counterpart of the proxy, it represents the client for the interface. The interface doesn’t know either whether it is communicating with the actual client or a stub. Although it is not totally true that the component implementation is independent from the location of the client, the stub pattern still helps keeping code identical for the implementation of the interface themselves. The implementation of a component will still be different whether it is

an in-process or out-of-process component since it will have to be a DLL in one case and a executable in the other. The design of the interfaces might also differ since out-of-process servers will tend to avoid too many round trips.

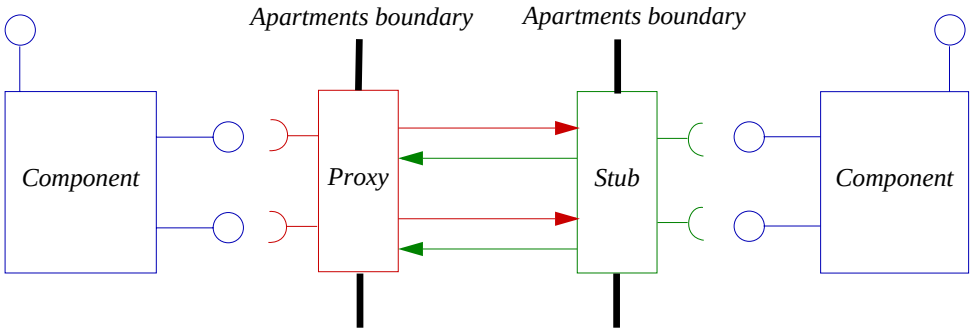

Figure 3: Cross Apartment Calls

There is one proxy/stub pair per interface. The proxy or the stub is loaded dynamically only when needed. This proxy/stub pair constitute the *marshaller*. The reason for having a single name for two different things come from how MIDL generates its code. MIDL will produce files for one DLL in which both the proxy and the stub will be included. This DLL is the marshaller.

Summary

This very brief introduction to the Component Object Model should be enough to get started with the EiffelCOM wizard. It specifies the main characteristics that define the type of a component and that need to be given to the wizard along with the definition file.