Delphi Meets COM: Part 1

by Dave Jewell

COM, or Microsoft's *Component Object Model*, is the foundation upon which an increasing amount of Windows technology is based. You will no doubt be aware that ActiveX controls are implemented as COM objects. OLE technology is also built around COM and much of the functionality of the Windows Explorer shell is only accessible through COM interfaces. Microsoft's Internet Explorer is itself built up from reusable, COM based components and even the friendly face of Office97 is provided by courtesy of a set of COM objects residing in proprietary Microsoft DLLs.

In this series on COM technology, Steve Teixeira and mysel will be looking at what COM actually is, describing how you can interact with COM components (with Delphi as the primary programming tool) and examining many practical applications such as shell extensions, ActiveX components and more. I'll start off and Steve will come in later in the series.

First of all, to gain a solid understanding of COM, we need to familiarise ourselves with the COM philosophy: the concepts upon which COM itself is based. That's the focus of this introductory article which places COM in its historical context, explains the basics of how COM works and the unique benefits which it brings. This is quite a lengthy introduction (the real meat starts next month!) but I'd strongly encourage you to read it carefully. To fully appreciate the versatility of COM, it's important to grasp how and why it came into being in the first place.

Why Microsoft Needed COM...

Microsoft's work on COM evolved out of *necessity* from the company's earlier work on OLE, what was once known as their Object Linking and Embedding technology. OLE was (and still is!) a system which enables one application to work with data that is created and managed by a completely separate application. For example, consider the classical example of placing an Excel spreadsheet into a Word document. In OLE terms, the spreadsheet is *embedded* in the Word document. The enclosing document is known as a *compound document* because it incorporates data from more than one application. In this example, the enclosing application, Word, is known as the *OLE container* because it contains data from another application.

When Microsoft first designed OLE, the critically important thing was that the data had to be live. So, for example, if any change takes place in the original Excel spreadsheet data, then any open compound document in which that spreadsheet is embedded must immediately reflect the change. This allows OLE technology to be used, for example, in financial markets, where share information is often being continually updated. The idea of having live embedded data is very intuitive from the end-user's perspective, but from a programming perspective it was very difficult to achieve. It should be obvious that OLE necessitates a lot of co-operative interaction between the OLE container and the *server application* (the application which originally created the embedded object).

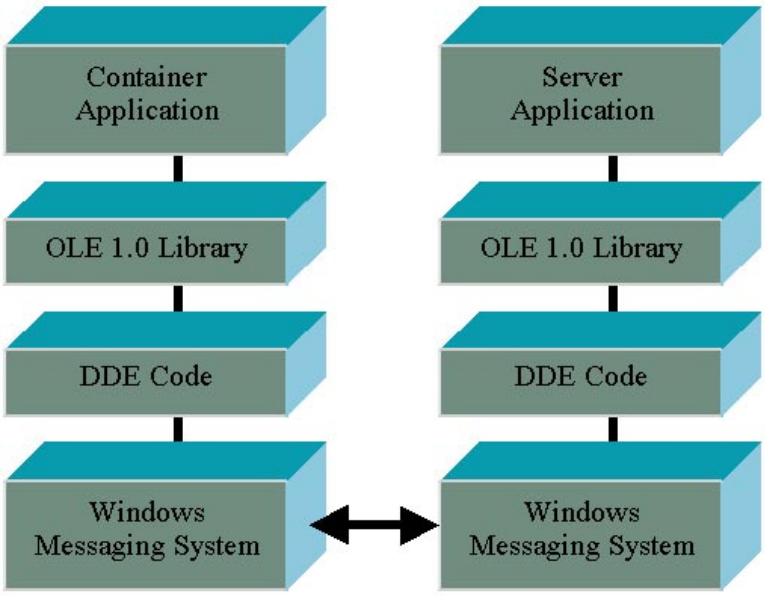
Clearly, Word does not know how to interpret Excel data or draw Excel spreadsheets: the container application needs to notify the server whenever all or part of an embedded object needs to be redrawn, and it must tell the server what part of the display area should be redrawn. Similarly, if our embedded spreadsheet is updated (possibly even over a network), there must be a mechanism for notifying the container of any changes.

Even the simple act of saving a compound document requires co-operation between server and container. Word does not know how to save an Excel spreadsheet as part of a Word document, it's the server application (Excel itself) which must stream the necessary data into the compound document at the appropriate point. From all this we can see that OLE necessitates a large amount of two-way conversation between the server and container applications, and the problem is obviously compounded (pun strictly intentional!) if a compound document contains embedded objects from many different applications.

Necessity, The Mother Of Invention

Earlier, I said that COM evolved out of necessity and that's absolutely true. Microsoft's earliest implementation of OLE, version 1.0, quickly gained a reputation as being horrendously slow and cumbersome. Want to slow your machine down to a crawl, or even a standstill? Just do something that involves an embedded OLE object and off you go... or not, as the case may be. OLE 1.0 was woefully inadequate for one simple reason: it was built on top of the (now mercifully obsolete) DDE protocol. DDE, or Dynamic Data Exchange was a scheme for allowing different applications to communicate with one another using the ordinary Windows messaging system. As you can imagine, this wasn't a recipe for blistering performance, see Figure 1.

DDE was easy to implement, but very slow in operation, it even used timeouts to detect whether or not the other end of the DDE connection had died! Not only was the whole thing an exercise in slow motion, but regretfully it wasn't particularly robust either. Even before OLE 1.0 was shipping, Microsoft's engineers realised that they needed a mechanism that was radically different and much more



➤ *Figure 1: Here's why Microsoft's initial implementation of OLE was no speed demon. Built on top of the slow and unreliable DDE technology, OLE quickly gained an unenviable reputation. This illustration demonstrates why it is that, sooner rather than*

efficient to communicate between different applications. The result of their deliberations was COM, a system much better suited to meet the increasing demands of the OLE architecture.

Did I say increasing demands? Oh, yes. While working on the underlying inter-process communication problem, Microsoft's engineers were simultaneously forging ahead with ambitious enhancements to OLE itself. With OLE 2.0, it became possible to edit an embedded document in place. Effectively, the server application temporarily 'takes over' the container window, adding its own menus, toolbars and so forth. In order to make this work, the amount of container and server interaction is greatly increased. For example, if you double click an embedded Microsoft Equation object in your Word document, you'll see Word's menus instantly replaced with those of the Equation server. Whenever in-place editing session is started, the server must negotiate with the container regarding the placement of menus and so forth.

From the above discussion, it should be obvious that COM 2.0 simply would *not* have been possible without the introduction of COM technology, thus dispensing with DDE and providing a more efficient mechanism for different applications to communicate with one another.

VBX Controls: Gone, And Best Forgotten...

While all this OLE development was going on, Microsoft were also contemplating a suitable successor to Visual Basic's 16-bit custom control technology.

As you may be aware, VBX files were a specialised type of DLL that could be accessed at design-time by the Visual Basic design environment, and at run-time by the running application. However, the design of these controls was unashamedly 16-bit: VBX controls used internal 16-bit pointers and offsets which made it very difficult to author them using compilers that only understood 32-bit pointers, Delphi and Turbo Pascal being prime examples. The internal architecture of VBX controls was also, it has to be said, somewhat baroque. Something different was needed for the brave new world of 32-bit programming.

At an early stage in the development of OLE 2.0, Microsoft's engineers realised that COM represented a much more general mechanism than was required by OLE alone. It could kill two birds with one stone: providing fast inter-process communication for OLE, while also permitting an OLE-based control to efficiently interact with a container application or design environment. They quickly came to appreciate that just as OLE allowed one application to seemingly work with data belonging to another program, so it was possible to embed any arbitrary type of control into a containing application. Just as a server application could notify a container application that a spreadsheet had been updated, a control could use a similar mechanism to (for example) tell the containing application that it had been clicked by a mouse. Thus was born the idea of OLE controls.

Although this was a great leap forward from a technical point of view, it represented something of a marketing disaster. Largely because of the OLE 1.0 fiasco, OLE was still perceived as being enormously complicated, difficult to program, buggy and cumbersome to use. Surely Microsoft weren't planning to create custom controls using the same technology? Surely not! Once the marketing department woke up to the problem, OLE controls were quickly re-christened *OCX Controls* in order to put some distance between the original DDE-based system, and the newer COM-based technology. More recently, Microsoft's control technology has again been re-christened, emerging as *ActiveX*.

Note that this shouldn't be taken as implying that the three terms are completely synonymous, that would be something of an over-simplification. Microsoft have continued to refine COM technology and the emergence of the Internet has meant that being able to create small, easily downloadable (so-called 'light weight') controls is now very important. Modern ActiveX components are very

much smaller and less complex than the original OLE controls.

The COM Advantage...

A moment's thought should convince you that designing an interface to an embeddable control is potentially a very complex business. How does Visual Basic know what properties to display when a particular control is selected? How can you work with an arbitrary ActiveX control inside Visual C++ or Delphi? The design environment cannot know, in advance, how many methods, properties and events the control will export. It doesn't know how to trigger events, or be triggered by them. It doesn't know what parameters should be passed in event notifiers, or what parameters are expected by each method call. How do you provide all this sort of information to the design environment? The answer, of course, is that general purpose controls must be able to 'advertise' their interface, just as an embedded OLE object needs to be able to communicate certain aspects of its behaviour to the container application.

Microsoft realised that one of the key requirements of COM was the ability to 'discover' a software interface programmatically at run time. In other words, it must be possible to query a software component in order to determine what its programming interface is. This is perhaps the most fundamental characteristic of the COM architecture.

Contrast this with the traditional DLL-based approach. If a client program knows that a given set of routines exist in a DLL, and it knows what parameters are required by those routines, then it can make use of the DLL. Conversely, if the client program (or rather, the author of the client software) doesn't have access to the DLL interface specification, then it's impossible to work with the DLL. What it really boils down to is this: *DLLs don't include the run-time information required to make use of them*. It's a bit like being handed a box of unfamiliar tools: you're told that the tools are all very useful, but you've got no information on what each tool is or how to use it.

Another significant advantage of COM is the fact that it's a language-neutral technology. This is just a fancy way of saying that you can use COM from any programming language whose compiler understands how to call a COM interface. This includes Visual C++ 5.0, Delphi 3.0 and Visual Basic 5.0. It also includes (to the annoyance of Java's portability purists!) Microsoft's Visual J++ development system, which overloads certain Java language constructs to provide direct support for COM programming.

Let's summarise the COM benefits we've discussed so far:

- COM provides a fast, efficient basis for different software modules (by which I mean applications, controls, services, or whatever) to communicate with one another.
- COM enables the programming interface to a module to be interrogated programmatically, thus making COM a suitable foundation for the building of reusable components and controls.
- COM is language neutral, it will work with any programming language that understands COM interfaces and, if you're feeling suitably masochistic, it's even possible to call COM interfaces directly from assembler code.

This isn't an exhaustive list of COM benefits by any means. For example, the introduction of DCOM (Distributed COM) means that client software on one machine can make use of software services running on another machine in a network. DCOM hides the fact that the COM object isn't local, it's accessed just as if it were local by the client code.

So What's An Interface?

Up to now, I've used the word 'interface' a number of times in a very loose sense: I haven't given a clear definition of what I mean by the word. In the COM sense, *an interface can be defined as a set of related routines that are grouped together into one logical entity*. You could argue that the Windows API is an 'interface' in this sense, because it defines a set of routines (thousands of them!) which relate to Windows programming. However, routines in a COM interface are normally related to one another much more strongly than that. If you imagine the Windows API divided up into functional groups, we might have one group devoted to memory management, another group for file I/O, a third for window management and so on. In principle, API routines grouped together in this way could be divided up into a number of COM interfaces, where each interface corresponds to a particular functional group.

All the routines in a single COM interface can be easily accessed once you've got yourself an *interface pointer* for that interface. An interface pointer is a 32-bit quantity which you can think of as a 'magic cookie' through which you access the various routines of the interface. If this sounds vague, then just think of things in Delphi programming terms: if I hand you a 32-bit pointer to a `TMemo` component, you can immediately start calling that component's methods.

The fact is, this analogy is a lot closer than you might think. If you're familiar with the 'nuts and bolts' of Delphi objects (I recommend the excellent *Delphi Component Design* by Danny Thorpe, Addison-Wesley, ISBN: 0-201-46136-6) then you'll know that every type of Delphi object has an associated VMT or Virtual Method Table. The VMT is really just an array of pointers to the different virtual methods supported by the object. The Delphi compiler knows what array index to use for each virtual method. If you understand this then, take heart, it's pretty much the same story with COM. At it's simplest, an interface pointer is essentially a pointer to an array of method pointers just as with COM.

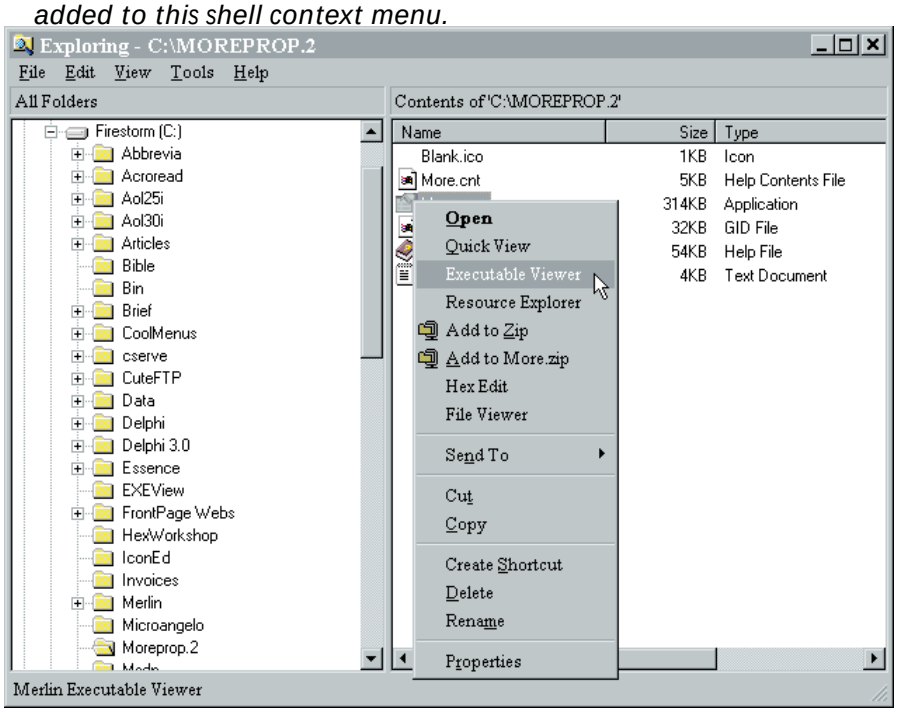
I said earlier that it was possible to interrogate a COM object programmatically to discover what programming interface it supports. But, how do you

communicate with the COM object in the first place if you don't know what interface is there? To resolve this chicken-and-egg situation, all COM objects must support a special interface called `IUnknown`. By convention, a COM interface name always begins with a capital 'I', just as you always use a capital 'T' to begin a Delphi type name.

Once you start communicating with a COM object through the `IUnknown` interface, you can easily get the object to tell you what other interface(s) it supports. What, more than one interface? That's right, an important characteristic of the COM architecture is that multiple interfaces can be supported by a single COM object. Most COM objects support at least two interfaces: `IUnknown` and the real interface that provides the 'business-end' functionality of the component. You can think of `IUnknown` as being a 'gateway' to any other interfaces supported by the COM object. Having obtained a pointer to `IUnknown`, the client software can then use this to retrieve a pointer to the interface its *really* interested in. It's for this reason that, in order to qualify as a fully paid up member of the COM community, every valid COM object absolutely *must* implement the `IUnknown` interface. You also need to appreciate that *every other interface provided by a COM object inherits from* `IUnknown`.

Note that since you're coming from an object-oriented Delphi background, you'll be familiar with the idea of inheritance where a class 'inherits' all the functionality of an ancestor class. In C++ or Delphi, a derived class inherits all the methods of the ancestor class together with any member fields that might be defined. This is an example of *code inheritance*, calling methods of a derived object will often cause code to be executed in methods of the ancestor class. COM does not support this type of inheritance. It is not possible to take one COM object, directly derive another object from it and expect the original object to be called as and when appropriate. Instead, COM offers *interface inheritance*. With interface inheritance, you simply inherit the 'specification' of how that object behaves including all the methods defined in the ancestor interface.

➤ *Figure 2: Increasingly, more and more Windows services are only accessible via COM. If you want to have a meaningful relationship with the Windows shell, then you need to speak COM. Here, you can see the various menu entries that Merlin and WinZIP have added to this shell context menu.*



Why is that every COM interface has to derive from `IUnknown`? The answer is polymorphism, the ability to treat the same object as though it were several different things. Because every COM interface includes the methods needed by `IUnknown`, we can quite legitimately pass *any* interface pointer to *any* routine that expects an `IUnknown` interface, and everything will work as advertised. Just as importantly, because all interfaces include these three methods (and notably, the `QueryInterface` call which we'll be looking at soon) it's very easy to find our way around a COM object no matter what interface pointer we currently have. As we shall see, the `QueryInterface` method is the all-important call for retrieving other interfaces of a COM object. If the `QueryInterface` method *wasn't* present in every interface then we'd have to find our way back to the original `IUnknown` interface before going in search of another interface. If this isn't clear to you now, it will be after I've explained how `QueryInterface` works.

One final remark about polymorphism in the context of COM: just as you can pass any interface to a routine that expects a `IUnknown` interface, in the same way, you can pass different implementations of the same interface to the same routine. To put this another way, suppose we had a hypothetical COM module such as a spell-checker module. Imagine that our spell-checker module implemented a spell-checking interface called `ISpellCheck`. By providing them with the documentation for this interface, many different manufacturers would be able to create different implementations of this interface, and the same application would be able to work with them all.

Even if some manufacturers created a new enhanced interface, existing applications would still work provided that the new interface inherited from `ISpellCheck`. This may seem like an obvious point, but it's obviously crucial to COM's interoperability. After all, the only reason you can plug an

unknown ActiveX control into a Web browser or a Delphi application is because the control implements the documented interfaces needed to function in such an environment. It would be a pretty poor browser that was 'hard-wired' to recognise only certain controls!

IUnknown: Mother Of All Interfaces...

OK, enough abstract theory. Let's begin to put things in concrete terms. The `IUnknown` interface contains exactly three methods. These are `AddRef`, `Release` and `QueryInterface`. Any implementation of `IUnknown` must provide these three methods and, because all other COM interfaces inherit from `IUnknown`, any other interface *must* likewise provide these three methods.

The code snippet in Listing 1 shows the Delphi 3.0 definition of `IUnknown`, taken from the SYSTEM.PAS file.

For now, don't worry about the gobbledy-gook on the second line. That's an example of a GUID or CLSID, something that I'll be describing soon. The most important of the three `IUnknown` methods is `QueryInterface`. It's through this method that you can ask a COM object whether or not it supports a particular interface. If it does, then it will give you back a pointer to the interface. If not, then a `Nil` value is returned and an error code is reported. `QueryInterface` returns an error code through the function result.

As the first parameter to `QueryInterface`, you specify `IID`, the ID of the interface that you are interested in. As you can see from the above, the type of this parameter is `TGUID`: it's another GUID.

For now, it's easiest to think of a GUID as a large number that is used to uniquely identify a particular interface. When you use ready-made COM objects, the programming documentation will include a list of these identifiers. For example, Borland's SHLOBJ.PAS file defines a GUID called `IID_IShellFolder`. This is one of the COM-based interfaces which is used to communicate with the Windows Explorer shell. Again, be patient for now, I'll be discussing GUIDs in more detail shortly.

```
IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

➤ *Listing 1*

```
{ Connect an IConnectionPoint interface }
procedure InterfaceConnect(const Source: IUnknown; const IID: TIID;
  const Sink: IUnknown; var Connection: Longint);
var
  CPC: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Connection := 0;
  if Source.QueryInterface(IConnectionPointContainer, CPC) >= 0 then
    if CPC.FindConnectionPoint(IID, CP) >= 0 then
      CP.Advise(Sink, Connection);
end;
```

➤ *Listing 2*

The second parameter to `QueryInterface` is an `out` parameter. This is new Object Pascal syntax that was introduced with Delphi 3 to support COM programming. For now, it's easiest to think of an `out` parameter as being similar to a `var` parameter, its value is changed as a result of the function call. `QueryInterface` examines the interface ID, `IID`, determines if that interface is supported by the COM object, and if so, returns an interface pointer for the object in the `Obj` parameter. If the specified interface is supported, then zero is returned as the function result. If the interface isn't supported, then `E_NoInterface` (a constant defined in WINDOWS.PAS) is returned. Other function results are possible, including the notorious `E_Unexpected`. This indicates that something has gone deeply wrong with COM itself and we should abandon ship at the earliest opportunity! It's worth nothing that in COM, error codes are negative, so testing the function result for being greater than or equal to zero is a sufficient test for success.

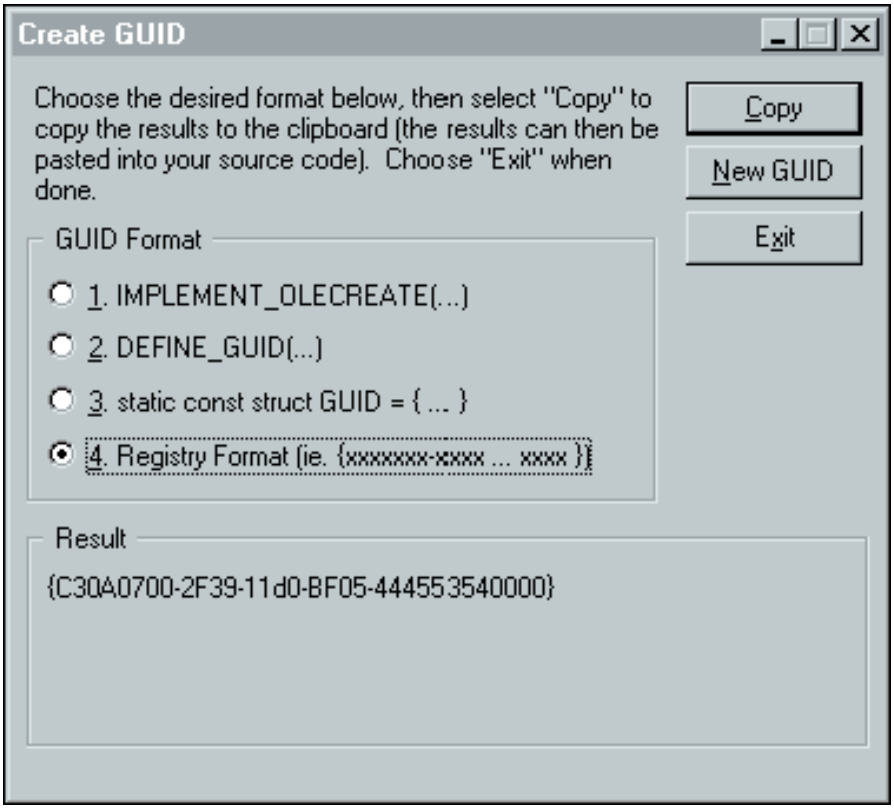
Just to give a flavour for using `QueryInterface` in the real world, here's another code snippet (Listing 2), this time taken from Borland's AXCTRLS.PAS file, part of the VCL source code.

I'm not too sure what this routine is doing and I don't much care! The important thing is to look carefully at the call to `QueryInterface` in the second line of code. The `QueryInterface` method is called on the `Source` object, which is of type `IUnknown`. If successful, it returns an interface pointer to a `IConnectionPointContainer` interface in the variable `CPC`. The `FindConnectionPoint` method of this new interface is immediately called and this, in turn, gives us a pointer to an `IConnectionPoint` interface in `CP`. Finally, the `Advise` method of this last interface is called. Wheels within wheels!

This routine is an excellent example of how you can 'find your way around' the different interfaces of a COM object using `QueryInterface`. In the above example, you shouldn't think of `CPC` as a different object to the original `Source` object. Both `Source` and `CPC` are interface pointers of the same object, *but each exposes a different interface*.

Incidentally, you should now appreciate why it is that all interfaces are derived from `IUnknown`. Suppose we've got a `IConnectionPointContainer` interface pointer for the `Source` object and we want to get to some other interface of the same object. Whatever interface pointer we've got, we can immediately call `QueryInterface` on it in order to get a pointer to the new interface. Without this capability, we'd have to always carry around a pointer to the object's `IUnknown` interface.

This will no doubt raise other questions in your mind such as how do we get an initial `IUnknown`



➤ *Figure 3: Here we can see GUIDGEN in action. This is a Microsoft utility that generates unique GUID numbers on demand. For Delphi developers, the equivalent functionality is already built into the Delphi 3 IDE. Just type Ctrl-Shift-G and off you go...*

pointer to an object? Again, this will become clearer as we work our way through the basics of COM programming: I've got to defer an explanation of that until we've discussed the idea of CLSIDs (class identifiers) and GUIDs. For now, just take it on trust that it can be done.

The `AddRef` and `Release` methods are simpler to use (they take no parameters), but very important: some COM objects might be implemented through a separate application while other COM objects might be implemented through a Windows DLL. Either way, the application or DLL needs to know when it can safely be unloaded from memory. It's not adequate for the client application to explicitly unload the COM object from memory since it might be in use by more than one client at the same time. It would obviously be pretty disastrous if Client A were to unload an in-use COM object from under the nose of Client B!

In order to address this problem, Microsoft introduced the idea of reference counting. If you're familiar with the Windows API, you'll know that this concept appears in various other places within Windows. For example, when an application wants to show or hide the cursor, it calls a routine called `ShowCursor`. If this routine is called multiple times, the system maintains an internal reference count to remember how many times the cursor has been hidden. The cursor will only reappear when the number of 'hide' calls is matched by the same number of 'show' calls. In the same way, the Windows memory manager has historically used routines such as `GlobalLock` and `GlobalUnlock` to lock and unlock memory blocks. As before, an internal reference count is used to ensure that a block isn't truly unlocked until the number of unlocks matches the number of locks.

It's just the same with COM objects. Many different clients can be using the same object, but none should need to know about the existence of the others and cannot therefore know whether or not a COM object should be released. Instead, the COM object *itself* maintains a reference count which is incremented each time a client calls `AddRef` and decremented whenever there's a call to `Release`. When a COM object receives a `Release` call which takes its reference count back down to zero, then it automatically destroys itself and is removed from memory.

This seems pretty straightforward, but there's one small wrinkle: it should be obvious that when a COM object first hands out a new interface pointer, it must *immediately* set the reference count to one. If it didn't do so, then it would be in use, but with a reference count of zero. To put this another way, a COM object can't realistically hand out an interface pointer and then sit around in memory waiting for the client to make an initial call to `AddRef`. Typically, the client wouldn't bother(!) and the whole raison d'être behind reference-counting would be defeated.

In the simplest scenario, a client would get a pointer to an interface, use it, and then call `Release` on that interface. It would never be necessary for the client to call `AddRef`. Why, then, did Microsoft bother to implement `AddRef` at all? The reason is that an application might sometimes need to pass a copy of an existing pointer on to another client. Under these circumstances, the COM object has suddenly acquired another client without being notified. The convention is that the new client (the one that receives the interface pointer from the existing client) must call `AddRef` to notify the COM object that there are now two outstanding references.

Note that a COM object can implement reference-counting on an interface by interface basis, or it can use one global reference counter for all its interfaces. Whether it makes sense to do things one way or another depends on a particular COM object and on the internal system resources needed to support a particular interface.

In Search Of Global Uniqueness...

By now, I expect you're frantic to know all about CLSIDs and GUIDs. If you're not, you should be! A GUID is essentially a very long number that occupies 16 bytes, or 128 bits. GUID stands for Globally Unique Identifier, terminology which is derived from the Open Software Foundation's somewhat presumptuous UUID or Universally Unique Identifier concept. The idea is that a GUID is so large that it can be used to uniquely identify an individual thing from out of a very large set of such things. A GUID isn't COM specific: you could use it to distinguish between books in the Library of Congress (like the ISBN number), between the shirts in your wardrobe, or anything else you liked.

A CLSID is simply a GUID that's used to identify a COM class. Thus, strictly speaking, all CLSIDs are GUIDs, but not all GUIDs are CLSIDs. The terms are often used interchangeably so don't panic if you see one where the other ought to be... It's important to emphasise that a CLSID doesn't identify a particular object (a specific instance of a class), but it identifies the class itself. Thus, in the aforementioned code fragment which defines the `IUnknown` class, we saw that `IUnknown` is identified by a CLSID of:

```
{00000000-0000-0000-C000-  00000000046}
```

When you create your own COM classes, you'll want to use a CLSID that nobody else has ever used. How can you ensure uniqueness?

Given that one CLSID uniquely identifies each COM class the world over, you might be forgiven for thinking that you have to go to Microsoft in order to be allocated a CLSID for your new COM object.

Well, it's certainly possible to do that. If you ask them nicely, Microsoft will be very happy to allocate you a chunk of 100 contiguous CLSIDs which are registered in the name of your organisation. But remarkably, it's possible to come up with your own CLSIDs and be very confident that nobody else has used the same CLSID for their use.

C/C++ programmers use a nifty little Microsoft-supplied utility called GUIDGEN. When you execute GUIDGEN, it will instantly generate a new GUID for you, present it in one of several different formats and allow you to copy it to the Windows clipboard for pasting straight into your development environment. As ever, things are even niftier for Delphi developers, the equivalent of GUIDGEN is built right into the Delphi 3 IDE! Simply hold down the `Control` and `Shift` keys, hit `G` and, hey presto, the IDE will dream up a new, unique GUID and paste it into the active code window.

Note that if you're like me, then you don't like taking things on trust and you're maybe thinking it's pretty unlikely that these auto-generated numbers are unique. However, Microsoft's programming documentation specifically states that GUIDGEN will never generate the same GUID no matter how many times it's run. If you're interested, it uses a number of cunning tricks to achieve numerical uniqueness. For one thing, it takes the current date and time into account, measuring the time to a high degree of accuracy. It also uses certain persistent state information to guard against the possibility of the clock being moved forwards or backwards. A 'forcibly--updated' counter also gets thrown into the mix, to ensure that numbers remain unique when GUIDGEN is called repeatedly at very frequent intervals. The software also looks for an installed network card and uses the unique network address of the host machine as another factor in generating the GUID! If no network card is present, then a pseudo machine-identifier is synthesised from a number of highly variable machine states including (but not limited to) RAM size, hard disk size, processor type and the installed hardware configuration. Although it *is* theoretically possible for the same GUID to be generated on two different machines, it's far more likely that you'll hit the Lottery jackpot for ten weeks consecutively! It would be rather nice, but I suspect the Sun will go nova first...

Summary

This month, I've looked at how and why COM came into being, listed some of the advantages it has over traditional DLL-based techniques, and explained the basics of the `IUnknown` interface. I've also described how CLSIDs are used to guarantee planet-wide global uniqueness between different COM classes.

COM is a complex subject and this has been a fairly lengthy introduction, but it's not over yet! Next month, I'll continue this series by giving you some more introductory material, and then go on to demonstrate how you can use COM to interact with the Windows shell courtesy of the COM extensions built into Delphi 3.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review*. You can contact Dave as Dave@HexManiac.com.