Delphi Meets COM: Part 2

by Dave Jewell

In last month's introductory article, I described some of the basic concepts behind Microsoft's versatile COM technology. This month, I'm going to continue with the introduction, add a little more flesh to the bones, and we'll then be well-placed to start writing some COM code.

The Conventional Way Of Depicting A COM Object

By convention, a COM object is represented as a box, the contents of which are not open to examination. Remember that a COM object is simply a 'black box' that provides one or more services. You should not know or care how the black box works internally, and the way you use that component should never be dependent on the internal implementation. COM objects communicate with the outside world solely through the interfaces that they provide.

When drawing a COM object, interfaces belonging to that object are drawn as a line extending from the object with a small circle at the end. As indicated in Figure 1, the `IUnknown` interface is always shown pointing vertically upwards, almost like the antenna on one of the Telly-Tubbies! That's because of the way in which it functions. As I explained last time, the `IUnknown` interface effectively fields interface enquiries, using the `QueryInterface` call to return a pointer to an interface which can do some real work. It should go without saying that the COM object in Figure 1 is of no use to man nor beast!

A more representative COM interface is shown in Figure 2. Here, we have a hypothetical COM object which provides language tool services. In this particular case, four different COM interfaces are exposed, in addition to the ever-present `IUnknown` interface. By convention, the 'real' interfaces which the object provides are drawn as extending horizontally from the object.

Remember: a single COM object may actually implement multiple interfaces. You mustn't think of a single interface as necessarily providing access to all of an object's functionality. Rather, as I've already stressed, it provides access to a set of functionally related routines, just like the four distinct interfaces in our language tools object.

Of GUIDs, IIDs And REFIIDs

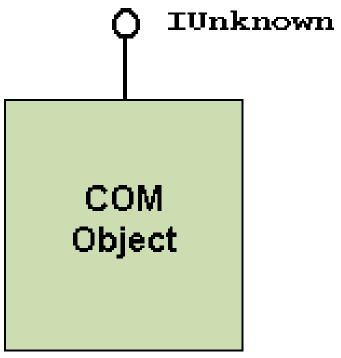
You'll remember that last time round, I introduced the idea of GUIDs, or globally unique identifiers. It turns out that a GUID is actually made up of four different fields, as shown in the declaration in Listing 1, which comes from SYSTEM.PAS.

There's not much real-world significance to the four different fields: it isn't like a TCP/IP address for example. I don't see why `TGUID` wasn't implemented simply as a straight array of 16 bytes: maybe this stuff is inherited from the Open Software Foundation who originated the concept. In any event, when you define a `TGUID`, you'll need to do something like Listing 2. This example is taken from the SHLOBJ.PAS file which ships with Delphi 3. It shows the GUID that's needed to communicate with the Windows Explorer's Shell Browser interface.

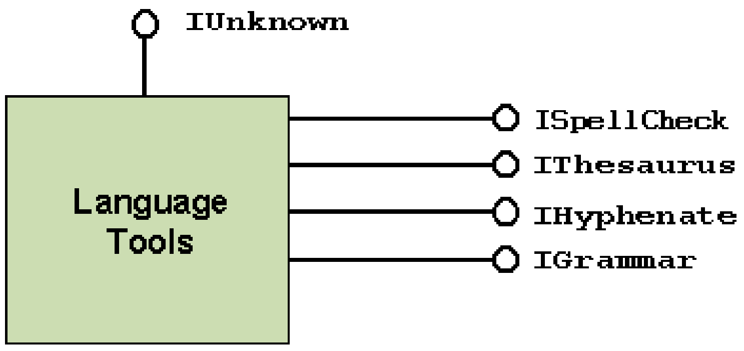
As a convenience, the Delphi 3 compiler will allow you to avoid all this D1, D2, D3, D4 nonsense by specifying a `TGUID` as if it were a string constant. This looks like a string constant, but it compiles to the same in-memory bit pattern as before (Listing 3).

I've already explained how the GUID which describes a class is called a CLSID or class ID. Unfortunately, that's by no means the end

➤ *Figure 1: The conventional way of depicting a COM object is with the IUnknown interface sticking straight up into the air like a receiving aerial; that's because it acts as a receiver for interface pointer requests.*



➤ *Figure 2: To be of any use, a real-world COM object must implement one or more useful interfaces. Here we see a hypothetical language tools component which provides four different types of service. Non-IUnknown interfaces are generally drawn as horizontal lines to the left or right.*



```
PGUID = ^TGUID;
TGUID = record
  D1: Integer;
  D2: Word;
  D3: Word;
  D4: array[0..7] of Byte;
end;
```

➤ *Listing 1*

```
const
  IID_IShellBrowser: TGUID =
    (  D1:$000214E2; D2:$0000; D3:$0000; D4:($C0,$00,$00,$00,$00,$00,$00,$46));
```

➤ *Listing 2*

```
Const
  IID_IShellBrowser: TGUID = '{000214E2-0000-0000-C000-000000000046}';
```

➤ *Listing 3*

```
const
  IID_IUnknown: TGUID = '{00000000-0000-0000-C000-000000000046}';
```

➤ *Listing 4*

of the story as far as COM jargon is concerned. Other types of entity you'll frequently encounter are the IID and the REFIID.

As well as being used for class identification, GUIDs are also used to identify interfaces to COM objects. An interface is identified using an IID or interface identifier: `IID_IShellBrowser` in Listings 2 and 3 is an example of an IID. However, because GUIDs are such big things (16 bytes) it's not convenient to pass them directly on the stack , at least, not in a language independent way. COM therefore introduces the idea of a REFIID which is essentially just a 32-bit pointer to an IID. Fortunately for us, Delphi will always pass a `const TGUID` parameter by reference, so we don't need to worry about REFIIDs to any great extent. I mention it here just in case you're saddled with a pile of C/C++ COM-related code and you need to figure out what's going on *[in which case you'd soon need the services of a psychiatrist! Editor]*.

Armed with the IID of a required interface, we can call `QueryInterface` to get a pointer to the interface we're after. COM defines a number of standard IIDs which you'll find in the OLE2.PAS file. One of these is `IID_IUnknown`, which is the identifier of the `IUnknown` interface itself (Listing 4).

If you're paying attention, you're probably feeling a bit puzzled at this point. After all, having once obtained a pointer to some interface on an object, why should you ever need to explicitly get back to the `IUnknown` interface? I've told you that any COM interface must inherit from `IUnknown`, which means that the interface pointer you've got already provides the `QueryInterface` method. You can use this method to instantly 'switch' over to any other supported interface. What then, is the point of going back to `IUnknown`?

It's a question of object equality. Suppose we're using our previously mentioned language tools object and we've got a pointer to an `ISpellCheck` interface and a pointer to an `IGrammar` interface. It might be important to determine whether or not the two interfaces refer to the same object. You can't just see if the two pointers are equal because you're comparing pointers to different method tables: there will never be equality. Instead you have to use `QueryInterface` to retrieve pointers to the two `IUnknown` interfaces and then do the comparison. For the purposes of testing object equality, a COM object is identified by the address of its `IUnknown` pointer. Listing 5 shows an example code fragment which calls `MessageBeep` if two pointers relate to the same COM object.

Of course, if you were going to do much equality testing, it would make sense to write a function to do this once rather than writing the code inline. Because of the inherent polymorphism in COM interfaces, you could define the function as taking two `IUnknown` pointers and then pass any type of interface pointers to the routine.

Delphi Meets COM

Now it's time to look in more detail at the COM support built into Delphi Pascal. In Listing 6 you can see the declaration for the `IMalloc` interface, which is defined in Borland's ACTIVEX.PAS file.

As you know, ordinary Delphi class types are prefixed with a `T` such as `TObject`, `TListBox` and so on. By contrast, interface declarations are prefixed with an `I`. Also, you'll see that this interface declaration uses the `interface` keyword rather than `class`. `IMalloc` is a built in system service that deals with memory management and, as you can see, `IMalloc` is defined as inheriting from `IUnknown`. As you'll recall from our discussion last month, this does not mean that `IMalloc` inherits any code from `IUnknown`. Rather, it indicates that `IMalloc` inherits the interface previously declared for `IUnknown`. Thus, if one were to look at the corresponding method table generated by the compiler, you'd see `QueryInterface`, `AddRef` and `ReleaseRef` occupying the first three 'slots.' Next would come `Alloc`, `Realloc` and so forth.

Note that as with ordinary Delphi virtual methods, it's important to stress that method table entries are assigned by the compiler in the order that they appear in the declaration. If you were to rearrange the six methods in `IMalloc`, putting them into alphabetical order, for example, then the code would compile perfectly but you'd end up with the wrong methods in the wrong method table entries.

This is a crucial point. COM interfaces are immutable: once defined and out 'in the field' they cannot be altered. If you want to enhance an interface, the correct approach is to add a new interface to your COM object so that older clients can fetch the old, existing interface, while newer, in-the-know, clients can get a pointer to the enhanced interface. In practice, if the new interface inherits from the old, then the first few method slots of the new interface will be occupied by the existing methods of the old interface. Thus, internally you can just pass the same interface pointer to both old and new clients irrespective of which IID they specify. Older clients won't know (and won't care) that there are more methods in the method table than they're aware of. I'm saying this because you might initially suppose that adding a new interface is a major hassle. It isn't.

I should also point out that when declaring interfaces, you can't

For Anoraks Only...

If, like me, you come from an assembler programming background (or are maybe just terminally inquisitive) then you might be interested in the code that Delphi generates when making COM calls. Look no further! I've never seen this discussed elsewhere, but I'm always happy to step in where angels fear to tread (keep quiet at the back there...).

The code below corresponds to the 'business end' of the equality test that I've just described. As with Delphi's long-string support, there is a certain amount of behind the scenes 'house-keeping' code associated with the use of COM interface pointers. In particular, there are two calls to `IntfClear` which initialise the `IUnknown` pointer variables before passing their addresses to the `QueryInterface` routine. I suspect that the compiler generates this initialisation code whenever it sees an interface pointer used as an `[out]` parameter to a COM call.

Particularly interesting is the call to `QueryInterface`. Because this method call always corresponds to the first entry in the COM object's method table, calling `QueryInterface` is very efficient. You'll also notice that , as with C++ and Delphi Pascal itself, calling a COM method involves an implicit `this` argument in addition to the method parameters which are directly visible to the programmer. See the code listing below:

```
:0040165B  lea   eax, dword ptr [ebp-0C]
:0040165E  Call  VCL30.System.@IntfClear      ; initialise u1
:00401663  push  eax
:00401664  push  00402010                     ; push address of IID_IUnknown
:00401669  mov   eax, dword ptr [ebp-04]
:0040166C  push  eax                          ; push implicit "this" (v1)
:0040166D  mov   eax, dword ptr [eax]         ; dereference v1
:0040166F  call  dword ptr [eax]              ; call QueryInterface
:00401671  test  eax, eax
:00401673  jle   0040169E                     ; check result
:00401675  lea   eax, dword ptr [ebp-10]      ; initialise u2
:00401678  Call  VCL30.System.@IntfClear
:0040167D  push  eax
:0040167E  push  00402010                     ; push address of IID_IUnknown
:00401683  mov   eax, dword ptr [ebp-08]
:00401686  push  eax                          ; push implicit "this" (v2)
:00401687  mov   eax, dword ptr [eax]         ; dereference v2
:00401689  call  dword ptr [eax]              ; call QueryInterface
:0040168B  test  eax, eax
:0040168D  jle   0040169E                     ; check result
:0040168F  mov   eax, dword ptr [ebp-0C]
:00401692  cmp   eax, dword ptr [ebp-10]      ; compare the pointers
:00401695  jne   0040169E
:00401697  push  00000000
:00401699  Call  user32.MessageBeep           ; it's beep time!
```

```
var
  v1: ISpellCheck;
  v2: IGrammar;
  u1, u2: IUnknown;
...
if v1.QueryInterface (IID_IUnknown, u1) > 0 then
  if v2.QueryInterface (IID_IUnknown, u2) > 0 then
    if u1 = u2 then MessageBeep (0);
```

➤ *Listing 5*

define member fields because a COM interface is inherently procedural. As with ordinary Delphi Pascal, you can define properties which simulate member fields, but the read and write specifiers for the property must resolve onto a procedure or function call rather than a member field. All methods of an interface are inherently public: the usual public, private, protected and published keywords are not allowed in an interface declaration.

As you can see from the declaration in Listing 6, all the methods are defined using the stdcall calling convention, which is the same convention you use when calling Windows API routines. This is the calling convention you should use when working with COM objects that can be called from other processes. However, there is another calling convention, safecall, which you use when implementing methods of dual interfaces. I haven't discussed dual interfaces yet, so you'll have to take that one on trust! Bear in mind that, as with ordinary Delphi class methods, the default calling convention of an interface method is register which is very unlikely to be what you want. You'd only use register methods when working with internal interfaces and objects which, of course, isn't COM programming at all.

```
IMalloc = interface(IUnknown)
  ['{00000002-0000-0000-C000-000000000046}']
  function Alloc (cb: Longint): Pointer; stdcall;
  function Realloc (pv: Pointer; cb: Longint): Pointer; stdcall;
  procedure Free (pv: Pointer); stdcall;
  function GetSize (pv: Pointer): Longint; stdcall;
  function DidAlloc (pv: Pointer): Integer; stdcall;
  procedure HeapMinimize; stdcall;
end;
```

➤ *Listing 6*

```
TComObject = class(TObject, IUnknown, ISupportErrorInfo)
private
  FRefCount: Integer;
  FFactory: TComObjectFactory;
  FController: Pointer;
  function GetController: IUnknown;
protected
  { IUnknown }
  function IUnknown.QueryInterface = ObjQueryInterface;
  function IUnknown._AddRef = ObjAddRef;
  function IUnknown._Release = ObjRelease;
  { IUnknown methods for other interfaces }
  function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
  { ISupportErrorInfo }
  function InterfaceSupportsErrorInfo(const iid: TIID): HResult; stdcall;
public
  constructor Create;
  constructor CreateAggregated(const Controller: IUnknown);
  constructor CreateFromFactory(Factory: TComObjectFactory;
    const Controller: IUnknown);
  destructor Destroy; override;
  procedure Initialize; virtual;
  function ObjAddRef: Integer; virtual; stdcall;
  function ObjQueryInterface(const IID: TGUID; out Obj): Integer;
     virtual; stdcall;
  function ObjRelease: Integer; virtual; stdcall;
  function SafeCallException(ExceptObject: TObject; ExceptAddr: Pointer):
    HResult; override;
  property Controller: IUnknown read GetController;
  property Factory: TComObjectFactory read FFactory;
  property RefCount: Integer read FRefCount;
end;
```

➤ *Listing 7*

Introducing TComObject

In order to do useful things, such as creating shell extensions, we need to get into a bit more detail at this point. The code in Listing 7 shows the class declaration for TComObject, an abstract class that defines the most important behaviour of a real-world COM object. Using TComObject we can, for example, create shell extensions, as you'll soon see.

If you read the Delphi documentation for TComObject, it states that:

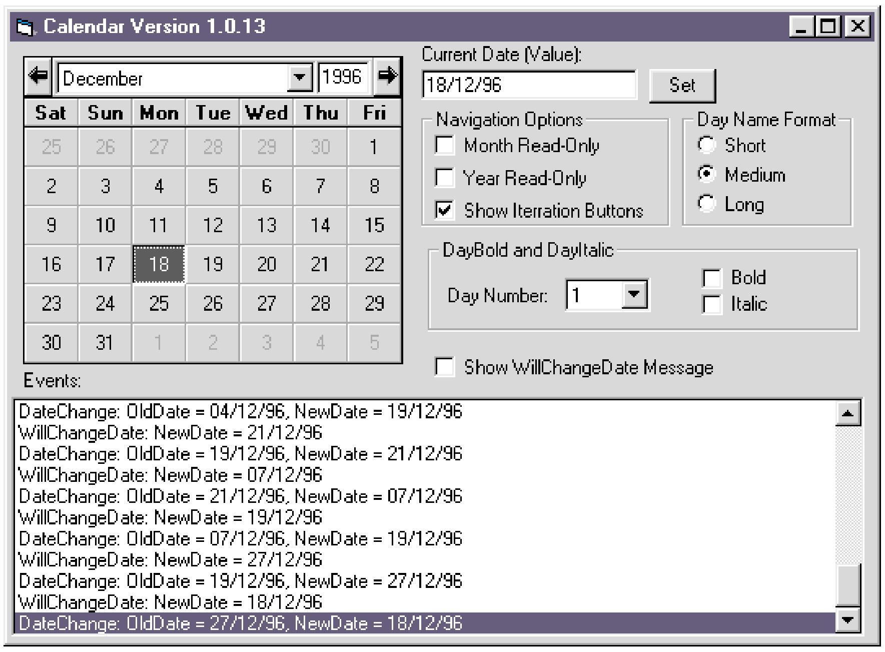
"TComObject has a CLSID and a class factory, so it can be registered and externally instantiated (instantiated from another module) ... TComObject also supports aggregation, OLE exception handling, and the safecall calling convention used for dual interfaces."

Oh dear, there's a lot of mysterious-sounding stuff there such as class factories, aggregation and dual interfaces that we haven't covered yet. Time for just a bit more theory before we get into the practice! But first, take a look at the declaration in Listing 7 for TComObject: there are a few more language features here that I need to explain.

Firstly, notice that TComObject isn't an abstract interface declaration: it uses the class keyword instead of interface. Rather, it's an honest to goodness class declaration. You should treat TComObject as an abstract class, deriving your own context menu handler (or whatever) from this object.

Secondly, take a closer look at the first line of the class declaration: in addition to indicating that TComObject is derived from TObject, this special syntax indicates that the new object is going to support one or more previously specified interfaces. In this case, the declaration indicates that TComObject will implement the IUnknown interface together with another interface called ISupportErrorInfo. Don't worry about the ISupportErrorInfo interface for now, suffice to say that this code is required by OLE Automation controllers which need to know whether or not a particular error object is going to be available. We'll be looking at this in more detail later in the series when we cover OLE Automation and ActiveX controls.

The declaration for TComObject also demonstrates another aspect of Delphi's COM-related language extensions: interface method mapping. Suppose you define a class which implements an interface called ISpellCheck. This spell-checking interface might include a method called AddCustomDictionary. By default, the compiler will look for a method called AddCustomDictionary in your new class.


Calendar Version 1.0.13

December 1996

Sat	Sun	Mon	Tue	Wed	Thu	Fri
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Current Date (Value): 18/12/96 Set

Navigation Options
- [ ] Month Read-Only
- [ ] Year Read-Only
- [✓] Show Iteration Buttons

Day Name Format: Short [✓ Medium] Long

DayBold and DayItalic
Day Number: 1
- [ ] Bold
- [ ] Italic

- [ ] Show WillChangeDate Message

Events:
```
DateChange: OldDate = 04/12/96, NewDate = 19/12/96
WillChangeDate: NewDate = 21/12/96
DateChange: OldDate = 19/12/96, NewDate = 21/12/96
WillChangeDate: NewDate = 07/12/96
DateChange: OldDate = 21/12/96, NewDate = 07/12/96
WillChangeDate: NewDate = 19/12/96
DateChange: OldDate = 07/12/96, NewDate = 19/12/96
WillChangeDate: NewDate = 27/12/96
DateChange: OldDate = 19/12/96, NewDate = 27/12/96
WillChangeDate: NewDate = 18/12/96
DateChange: OldDate = 27/12/96, NewDate = 18/12/96
```

➤ *Figure 3: Here's an excellent example of COM aggregation. This is one of the sample ActiveX controls that Microsoft provided along with the Visual Basic 5 CCE Edition. This calendar control is actually aggregated from several other components, including buttons, checkboxes, radio groups, and so forth. But it only exposes one COM object to the world and 'looks' like a single object.*

```
constructor TContextMenuObject.Create;
begin
  inherited Create;
  iContextMenu := TOwnedContextMenu.Create (self);
  iShellExtInit := TOwnedShellExtInit.Create (self);
  ...more code...
end;
```

➤ *Listing 8*

However, what if your COM object implements several different interfaces, some of which include methods with the same name? Or what if you merely want to give a more descriptive name to the method which indicates what interface it's associated with?

Fortunately, Mr Hejlsberg thought of that! If you look at the beginning of the `protected` section, you'll see an example of interface method mapping where the three `IUnknown` methods are mapped onto new method names, these being `ObjQueryInterface`, `ObjAddRef` and `ObjRelease`. When using interface method mapping, bear in mind that you can only map a method onto a new name within the same class: you can't map a method over onto the implementation part of a wholly different class. Also, the number and types of parameters, the function result and the calling convention must all be compatible with the original method declaration.

Delphi 3: Aggregation Without Aggravation!

If you've been keeping back copies of *The Delphi Magazine*, now would be a good time to go in search of Issue 15. If you look at my *Beating The System* column in that issue, you'll find that I present the code for a COM-based context menu handler written using Delphi. Because this article was written way back in 1996, I developed the code using Delphi 2, which doesn't have as many whizzy COM based language extensions as Delphi 3.

If you look at that old 1996 code (a year is a long time in this business!), you'll see that in order to interface properly with the Windows Explorer, a context menu handler has to implement two different interfaces, `IContextMenu` and `IShellExtInit`. Broadly speaking, `IShellExitInit` is responsible for retrieving the name(s) of the selected file(s) from Windows Explorer, whereas the `IContextMenu` deals with the business of adding custom menu items to the Explorer's context menu and receiving notifications when a custom menu item is called.

My original Delphi 2 code worked by declaring two new classes, `TOwnedContextMenu` which implemented the `IContextMenu` interface, together with `TOwnedShellExtInit` to implement the `IShellExtInit` interface. Another class, `TContextMenuObject`, was effectively the 'owner' of the two aforementioned classes. In the constructor of `TContextMenuObject`, I created a private instance of each of the two owned classes, as shown in Listing 8.

Similarly, these two owned objects were destroyed when `TContextMenuObject` was itself destroyed. Because the outer level class had to directly implement the two aforementioned classes, this involved a bit of necessary `jiggery-pokery` inside the `QueryInterface` code for the `TContextMenuObject` object. Listing 9 shows what it looked like.

As you can see, the `QueryInterface` code checks to see what interface is being requested, and returns a pointer to the appropriate object. With hindsight, this old code is a little confusing because the two internal objects are named with `i` as the first letter of their name, which makes them sound like abstract interfaces. However, what's being returned here are pointers to one of the two internal objects, a pointer to the object `TContextMenuObject` itself, or `Nil` if the supplied interface identifier isn't recognised.

This is a classic example of aggregation. From the viewpoint of the client software (the Explorer in the case of a context menu handler), `TContextMenuObject` appears as a single, monolithic object which supports all the needed interfaces. The fact that, internally, it actually comprises three objects is completely hidden. That's what we mean by aggregation: aggregating what appears to be a single COM object from a number of other, constituent, COM objects.

Aggregation is a powerful concept. As I mentioned earlier, Delphi 3's `TComObject` class provides direct support for aggregation: that's the purpose of the `Controller` property. If a `TComObject` is one of a group of 'inner' aggregated objects, then the `Controller` property will point to a 'master' `IUnknown` interface which is maintained by the overall 'super-object' (for want of a better phrase). In this case, the three standard `IUnknown` methods of all interfaces except `IUnknown` are routed through this controlling interface rather than being handled natively by the inner object. This is best demonstrated with a snippet of code from COMOBJ.PAS (Listing 10).

Unashamed Delphi zealot though I am, the best way to see aggregation in action is to play around with the custom control creation facilities of Visual Basic 5.0. With VB5, you can visually lay out what looks like a standard form, but is actually the display surface of a new, ActiveX super-component. You can then add existing COM objects to this 'form,' bind them all together with appropriate code, expose the necessary properties, methods and events to the outside world and, hey presto, you've got an aggregated object which is indistinguishable from a non-aggregated object as far as client software is concerned. The only difference is that it took a fraction of the time that it would have taken to build the entire thing from scratch. You see, reusable component based development can be applied on the micro level as well as the macro level...

Having said that aggregation is powerful, there's no point going to the extra effort of creating inner objects when you just don't need them. Using the COM support in Delphi 3.0, you can directly provide multiple interfaces on one

```
function TContextMenuObject.QueryInterface (const iid: TIID; var obj): HResult;
const
  { The interface ID's we can respond to }
  IID_IContextMenu : TGUID = (D1:$000214E4; D2:$0000; D3:$0000;
    D4:($C0,$00,$00,$00,$00,$00,$00,$46));
  IID_IShellExtInit: TGUID = (D1:$000214E8; D2:$0000; D3:$0000;
    D4:($C0,$00,$00,$00,$00,$00,$00,$46));
begin
  Result := 0;
  if IsEqualIID (iid, IID_IUnknown) then begin
    Pointer (obj) := self;            { Wants IUnknown - return self }
    AddRef;
  end else if IsEqualIID (iid, IID_IContextMenu) then begin
    Pointer (obj) := iContextMenu;    { Wants IContextMenu - return it }
    AddRef;
  end else if IsEqualIID (iid, IID_IShellExtInit) then begin
    Pointer (obj) := iShellExtInit;   { Wants IShellExtInit - return it }
    AddRef;
  end else begin
    Pointer (obj) := nil;
    Result := E_NoInterface;
  end;
end;
```

➤ *Listing 9*

```
function TComObject.QueryInterface (const IID: TGUID; out Obj): Integer;
begin
  if FController <> nil then
    Result := IUnknown(FController).QueryInterface(IID, Obj)
  else
    Result := ObjQueryInterface(IID, Obj);
end;
```

➤ *Listing 10*

```
TContextMenu = class(TComObject, IShellExtInit, IContextMenu)
private
  szFile: array[0..MAX_PATH] of Char;
public
  function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst, idCmdLast,
    uFlags: UINT): HResult; stdcall;
  function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;
  function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
    pszName: LPSTR; cchMax: UINT): HResult; stdcall;
  function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
    hKeyProgID: HKEY): HResult; stdcall;
end;
```

➤ *Listing 11*

COM object without the need for aggregation. The class declaration shown in Listing 11 is taken from one of Borland's COM examples.

As before, this is the declaration for a shell context menu handler, derived from `TComObject`. The difference here is that we can directly tell the Delphi 3 compiler to support `IShellExtInit` and `IContextMenu` within the one object. The four needed methods (three from `IContextMenu` and one from `IShellExtInit`) are implemented directly within the class. In fact, to implement your own context menu handler, pretty well all you need to do is provide the code for these four methods. This is a lot less work than was needed for my original Issue 15 event handler. In fact, Borland really couldn't have made it any simpler for you than they have done!

The Class Factory: Getting Into Manufacturing...

But wait, why did I say *pretty well* all you need to do? You're right, there is just one more wrinkle before we can dive in and start churning out millions of context menu handlers! We also need to understand a little about class factories.

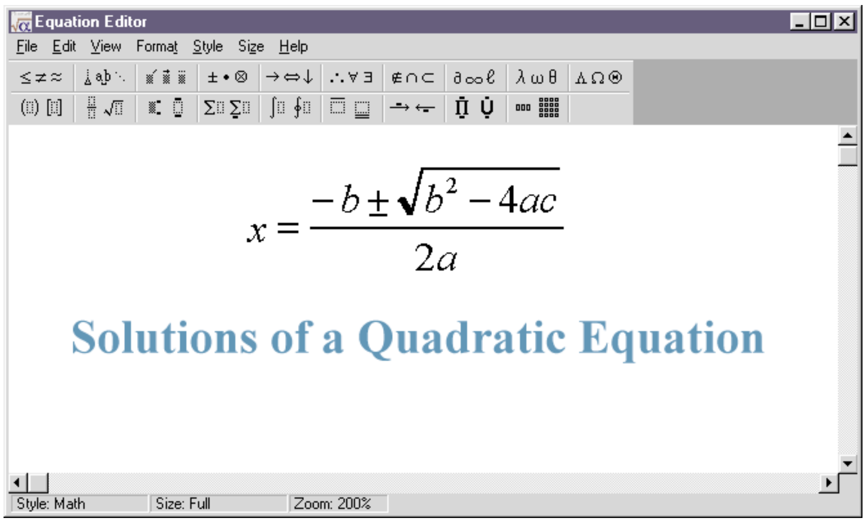
As the name suggests, a class factory is a special type of class whose only role in life is to create other classes. A class factory provides a standard interface for creating COM classes, irrespective of the vagaries of a particular class. When you create a new COM class, it's generally the responsibility of you, the COM class creator, to provide a class factory that can be used by the COM client to create new instances of the target class. Thus, when creating a context menu handler, the Windows Explorer does not actually create the TContextMenu class (Listing 11) directly. Instead, the server software (the DLL which is called by Explorer) must provide a class factory which is then used to create an instance of TContextMenu.

Interestingly, a class factory is itself a COM class but you'll be relieved to know that there are no such things as class factory factories! In other words, the client software can create the class factory directly and it then uses the class factory to create the target class. So why can't the client create the target class directly? As already stated, the class factory interface is designed to provide a uniform, consistent interface which insulates the client from having to worry about the details of constructor methods, the different arguments required by the constructor and so forth.

Of course, it is possible to instantiate target classes by making a direct call to the constructor. Whether or not this is appropriate really depends on the type of COM object that's being developed. In the case of context menu handlers, the Explorer expects to find a class factory.

Incidentally, up until now, we've been looking at COM classes largely from the perspective of the COM class author, but how does the client get an instance of a class factory and then use it to create an instance of the target class?

In the case of Explorer, everything happens through one simple Windows API call, namely CoCreateInstance. You pass the required CLSID to this routine and it searches the registry to discover the name of the server that implements the class. The server is loaded, a pointer to the class factory interface is obtained, and the CreateInstance method is invoked to obtain a pointer to the target class. A lot happens behind the scenes, but it's just one simple call as far as the client code is concerned.



$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Solutions of a Quadratic Equation

➤ *Figure 4: The Microsoft Equation Editor is an example of an out-of-process COM server. It's implemented as a separate EXE file, but provides COM services to interested parties. As to what a quadratic equation might be, that'll have to wait for another time...*

```
TComObjectFactory.Create(ComServer, TContextMenu,
  CLSID_ContextMenuShellExtension,'', 'Delphi 3.0 ContextMenu Example',
  ciMultiInstance);
```

➤ *Listing 12*

In next month's instalment of this series, we'll be looking in more detail about how the Windows registry fits into the 'big picture' and giving more examples of instantiating COM classes from the viewpoint of client software. In the meantime, if you go back to Issue 15 you'll find that my context menu example also, of necessity, included a class factory implementation. There wasn't a massive amount of code involved but it's comforting to note that, once more, the whole thing gets much simpler under Delphi 3.0. Listing 12 shows how to implement a class factory with Delphi 3.0.

There, that wasn't too difficult was it? Again, this code was taken from one of the Borland demos which ships with Delphi 3.0. This one-liner (assuming you have a wide screen!) creates a class factory to handle the instantiation of the TContextMenu class. Unlike an ordinary constructor, you don't need to do anything with the result of the above Create call. Behind the scenes, the new class factory is automatically added to a list of class factories managed by the COMSERV unit. It's this unit which implements the nuts and bolts code needed to roll your own COM servers, of which a context menu handler is a good example.

I've used the word 'server' several times now without precisely explaining it. As you've probably gathered, a COM server is a chunk of software that's responsible for implementing one or more COM objects and dishing out instances of those objects to any interested party that wants one. Most of the time, COM servers come in two flavours: in-process servers and out-of-process servers. This is really just COM jargon for DLLs and EXE files. A COM server that's packaged as a DLL is an in-process server because, like any Windows DLL, it lives in the same address space as the process that's using it. By contrast, a COM server that's implemented as an EXE file is an out-of-process server, because it lives in an address space all of its own.

Our shell context menu handler is an example of an in-process COM server. The DLL is loaded directly into Explorer's address space and it communicates directly with Explorer. On the other hand, if you wanted to make use of the built-in COM objects provided by Microsoft Word, then you'd be using an out-of-process server. Some out-of-process servers exist only to provide OLE services and can't be used as applications in their own right. Those ubiquitous ActiveX controls (OCX files) are essentially just DLL files that have been renamed to have a 'OCX' extension. ActiveX controls are therefore examples of in-process COM servers.

Until Next Time...

That about wraps it up for this month. This time round, we've worked through a lot of interesting material and you should now be sufficiently 'COM-cognisant' to understand much of what's involved in writing an in-process COM server such as a context menu handler. In next month's instalment, I'm going to give you one or two tasty examples of context menu handlers which are designed to make life easier for Delphi programmers. We'll also be looking at the role of the Windows registry in the COM scheme of things and discussing other new concepts such as dispatch interfaces, dual interfaces, OLE automation and more... See you then!

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