Delphi Meets COM: Part 5

by Dave Jewell

The shell extensions that we've looked at previously are examples of in-process COM servers. As promised last time round, this instalment is going to concentrate on OLE Automation. An OLE Automation server is an example of an out-of-process server because it's a separate application living in a different address space to the calling process. We'll begin by taking a look at the OLE Automation samples that Borland included with Delphi 3. Having dissected the code in these samples, we'll be better-placed to move on to something a little more adventurous.

If you open the directory DEMOS\OLEAUTO\AUTOSERV (relative to where you've installed Delphi 3) you'll find a couple of projects called AutoDemo and MemoEdit. The latter is a very simple OLE Automation server which implements an MDI-based text editor. It's the MemoEdit application which constitutes the server. AutoDemo is the controller application, you can see it running in Figure 1. Effectively, the AutoDemo application is a simple 'remote control' through which you can control various aspects of the server application such as creating a set of three blank memo windows, adding text to the windows, tiling and cascading the MDI children, and then removing the memo windows.

Nuts And Bolts Of OLE Controllers

Let's begin by taking a look at the code used to implement the controller, AutoDemo. Open the AutoForm.pas unit and scroll down to the TMainForm.FormCreate procedure. You'll see a line of code which looks like this:

```
MemoEdit := CoMemoApp.Create;
```

MemoEdit is a member variable of type IMemoApp and it's being initialised through a call to CoMemoApp.Create, but where are these other goodies defined? If you look at the uses clause for this application, you'll see a unit called Memo_TLB. The _TLB prefix indicates that this is a type library expressed in Pascal. If you open this unit in the usual way, you'll find that CoMemoApp is defined as follows:

```
CoMemoApp = class
  class function Create: IMemoApp;
  class function CreateRemote(const
    MachineName: string): IMemoApp;
end;
```

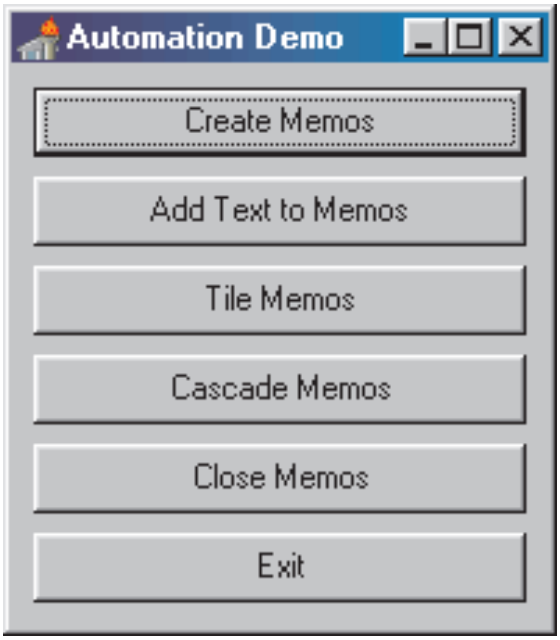
Because CoMemoApp.Create is a class function, we don't need to create an instance of this class before calling the function. This explains how we can immediately call the Create method in the controller code. Similarly, if you look through the rest of the type library file, you'll find the definition of the IMemoApp interface as shown in Listing 1.

```
IMemoApp = interface(IDispatch)
  ['{55E49D31-9FFE-11D0-8095-0020AF74DE39}']
  function NewMemo: OleVariant; safecall;
  function OpenMemo(const FileName: WideString): OleVariant; safecall;
  procedure TileWindows; safecall;
  procedure CascadeWindows; safecall;
  function Get_MemoCount: Integer; safecall;
  function Get_Memos(Index: Integer): OleVariant; safecall;
  property MemoCount: Integer read Get_MemoCount;
  property Memos[Index: Integer]: OleVariant read Get_Memos;
end;
```

➤ *Listing 1*

Thus, the call to CoMemApp.Create is effectively giving us a pointer to this interface. It's this interface which is 'exported' by the MemoEdit automation server, and the various methods in this interface define the 'API' of the class.

As pointed out in last month's instalment, the type library encapsulates the interfaces, methods and parameters that are associated with a particular COM/OLE server. The Memo_TLB unit is fundamentally a Pascal representation of the type library, but it isn't the type library itself. You shouldn't try to modify the Memo_TLB unit because it will automatically be regenerated by Delphi every time



➤ *Figure 1: Here's the small controller application from Borland's OLE Automation demo. The various buttons allow you to modify the edit windows of what's essentially an MDI-based notebook program.*

the type library is modified. That being so, will the real type library please stand up? If you look through the files in the project's directory you'll find a file called MEMOEDIT.TLB. This is the *actual* type library, it's copied into the OLE Automation server as a binary resource of type TYPELIB when Delphi rebuilds the server executable.

Ok, so we've got a type library, and Delphi has converted it into a human-readable (and compiler-readable!) Pascal representation. Having obtained an interface to IMemoApp, it's dead easy for the controller to 'pull the server's strings', so to speak. For example, when you click the controller's Create Memos button, this line of code gets executed:

```
for I := 1 to 3 do
  Memos[I] := MemoEdit.NewMemo;
```

This simply calls the NewMemo method to create three new MDI windows in the server. If we look at the definition for IMemoApp.NewMemo, we see that its return type is OLEVariant:.

```
function NewMemo: OleVariant;
  safecall;
```

Hmmm... so what's an OLEVariant? The simple answer is that it's an OLE-specific version of Delphi's built-in Variant type. As with plain vanilla Variants, an OLEVariant is an amorphous type that can be used to store objects of different types. However, an OLEVariant can only store data types that may be legitimately passed across an OLE Automation interface. This excludes certain native Pascal types such as PChar, Word and Pointer. As with an ordinary Variant, OLEVariant is more sophisticated than a simple variant record type. In particular, it remembers the type of object which it contains and, in effect, you can perform type-specific operations on an object of type OLEVariant without the need for any intermediate typecasting.

To see just what I mean by this, take a look at Listing 2 which shows

```
procedure TMainForm.AddTextButtonClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 3 do
    if not VarIsEmpty(Memos[I]) then
      Memos[I].Insert('This text was added through OLE Automation'#13#10);
end;
```

➤ *Listing 2*

the TMainForm.AddTextButtonClick method.

For each element of the three-item array, the standard VarIsEmpty function is called to determine if the array element is assigned. If it is, the Insert method is called to add text into the designated memo window. Where on earth did this Insert method come from? If you look back in the Memo_TLB unit, you'll see that there's another interface defined: IMemoDoc, which encapsulates the document interface for the MemoEdit server and (surprise, surprise!) includes a method called Insert. It should be obvious from this that the OLEVariant type returned by calling NewMemo is actually an IMemoDoc interface. The reason we didn't need to cast the array item to an IMemoDoc interface is because the OLEVariant already 'knows' what type of automation object it contains.

Late Binding Versus Early Binding

Actually, the above isn't strictly accurate. It's perhaps more accurate to say that the OLEVariant type doesn't know and doesn't care what data type it contains, and neither does the compiler! Confused? Think about it like this: when the method TMainForm.CreateButtonClick is compiled, the compiler sees that you're calling the NewMemo method of an IMemoApp interface. Consequently, it's able to generate code which directly references the NewMemo method in the vtable of the associated object. This is early binding because the compiler knows what the vtable looks like and which method is being called.

However, when you call the Insert method of some arbitrary OLEVariant object, the compiler hasn't got a clue. It doesn't know where the Insert method comes from (it can't reasonably assume it's the one referenced in the type library file) and even if it could uniquely identify the method being called, there's no way of verifying, at compile-time, that the object in question supports the method. This is where late binding comes into play. In these circumstances, late binding is the only possibility. To prove the point, try replacing the reference to the Insert method with this:

```
Memos[I].BambleWeeny(
  'This text was added through '+
  'OLE Automation'#13#10);
```

or even:

```
Memos[I].TotallyFictitiousMethod(
  0, 1, 2, 3);
```

You'll find that the compiler will accept these statements without a murmur. Once the compiler realises this is a late binding situation, it adopts a grovelling 'sure, whatever you say, boss' mentality! The only thing it will actively prevent is the passing of a non-automation compatible argument such as PChar or Pointer.

Of course, type-checking and linkage hasn't been eliminated, it's only been deferred. When you try running the program and the offending method call is executed, the Delphi runtime system will raise an exception with the message *'Method 'BambleWeeny' not supported by automation object'* or words to that effect. Similarly, if you refer to an existing method but enter the wrong number of arguments, or arguments of the wrong type, then again, an exception will be raised.

Note that in certain circumstances, passing the wrong argument type will be allowed because of the implicit type conversions

that take place. For example, zero or 3.1415926 will be happily accepted as numeric arguments to the Insert method, whereas what was really expected was a WideString. Rather than raising an error, the runtime code will do its best to reconcile any differences of opinion by converting the numeric argument into its alphanumeric representation.

The Best Of Both Worlds...

Incidentally, if you've been following the above discussion carefully, you'll appreciate that the interface pointers in OLE Automation allow both compile-time vtable binding (early binding) and runtime dynamic binding through a dispinterface. OLE Automation interfaces are *dual interfaces*, a concept that I haven't previously discussed. It's easy to understand how a dual interface works: because it derives from IDispatch, it implements Invoke, GetIDsOfNames and the other IDispatch interfaces that we talked about last month. However, the various IDispatch methods only take up a few slots in the vtable and, following these methods, we can have any number of regular methods that are accessed through direct vtable indirection.

This type of interface gives us the best of both worlds. Simple interpreted languages such as Visual Basic can access OLE Automation objects by using the dispinterface part of the interface, calling Invoke to reference the required methods. On the other hand, vtable-aware systems such as Visual C++ and Delphi can 'hit the metal' directly, accessing methods by early binding through the vtable. I specifically chose Borland's AUTOSERV demo because it's an excellent demonstration of how both early and late binding are often used even within the same controller application.

For the terminally curious, this leaves one interesting question. Namely, in the case of late binding, what code gets generated by the compiler and how is this used to reference the target method at runtime? To answer that question, take a look at Listing 3, which represents a sort of 'Anorak's Guide to the Delphi Late Binding Implementation.' This particular code snippet was excerpted from the AutoDemo controller application and corresponds to part of the for loop where the Insert method is called for each of the three OLEVariant items in the array.

Within the for loop, the VarIsEmpty routine is called to determine if a particular element of the array is assigned. If so, then a special runtime library routine called DispInvoke is called to make the actual late binding method call. DispInvoke is defined as being a cdecl routine, which means that the parameters are pushed onto the stack in right to left order. Thus, when looking at the code listing, the first parameter is the last item to be pushed on the stack, and the last parameter is pushed first! With this in mind, you'll see that DispInvoke takes four parameters. The first thing pushed (final parameter!) is a series of items corresponding to the arguments to the method. In this case, there's just one argument: the string that's being passed to Insert.

Note that the Delphi Pascal compiler doesn't bother to generate wide strings on the fly when passing string literals to OLE Automation methods that expect a WideString. Instead, the conversion to WideString format is performed inside the runtime library code.

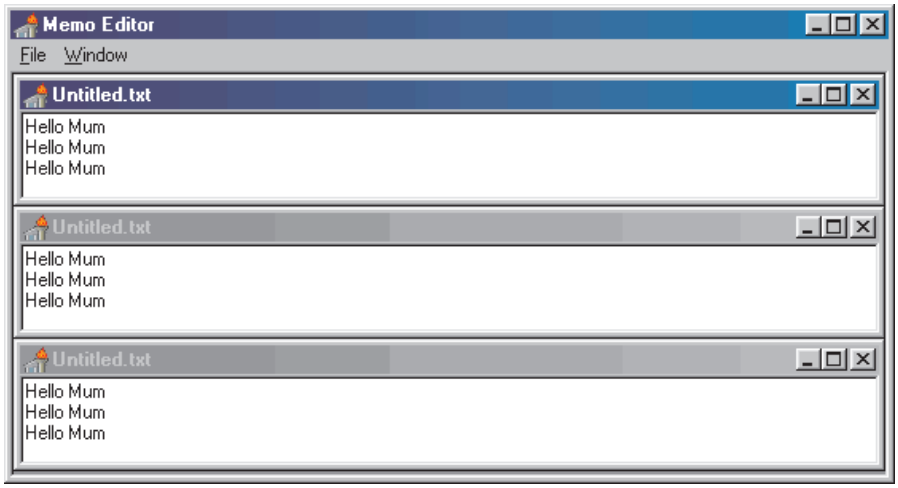
The next thing pushed (third parameter!) is a pointer to a special data structure called the call descriptor: more on this in a moment. Next, DispInvoke expects a pointer to the OLEVariant item itself. Finally (remember, this is the first parameter), there's a pointer to where the method result is going to be stashed. Since this is a procedure call, it's a Nil pointer that gets pushed onto the stack. It's important that DispInvoke is defined as a cdecl routine because of the variable number of method arguments that are pushed on the stack. As every good C/C++ programmer knows, the C calling convention dictates that the caller performs the stack cleanup which is what you can see happening at location $00401B37.

The heart of this implementation is the call descriptor. In Listing 3, the call descriptor is located at $00401B7C and so a pointer to this address is what gets pushed as an argument to DispInvoke. The call descriptor encapsulates the name of the method that is called (can you spot the ASCII string Insert in Listing 3?) together with an indication of the number of parameters passed, and their various types.

A full understanding of what's going on here is left as an exercise to the dedicated anorak. If you want to pursue this further, the relevant code can be found in Delphi's COMOBJ.PAS source file, where a DispInvoke call winds its way down onto a call to the deeply inscrutable DispatchInvoke procedure which, as you might expect,

➤ *Listing 3*

```
:00401B0E 8D84FEE8010000          lea eax, dword ptr [esi+8*edi+000001E8]
:00401B15 E856F5FFFF              Call System.VarIsEmpty
:00401B1A 84C0                    test al, al
:00401B1C 751C                    jne 00401B3A
* StringData Ref from Code Obj ->"This text was added through OLE Automation"
:00401B1E 684C1B4000              push 00401B4C
:00401B23 687C1B4000              push 00401B7C
:00401B28 8D84FEE8010000          lea eax, dword ptr [esi+8*edi+000001E8]
:00401B2F 50                      push eax
:00401B30 6A00                    push 00000000
:00401B32 E819F5FFFF              Call System.@DispInvoke
:00401B37 83C410                  add esp, 00000010
:00401B3A
// Code deleted for the sake of brevity...
:00401B7C 01                      add byte ptr [ecx], al
:00401B7D 0100                    add dword ptr [eax], eax
:00401B7F 48                      dec eax
:00401B80 49                      dec ecx
:00401B81 6E                      outsb
:00401B82 7365                    jnb 00401BE9
:00401B84 7274                    jb 00401BFA
:00401B86 0000                    add byte ptr [eax], al
```



➤ *Figure 2: And here's the Automation server itself. Because it shares the MEMO_TLB.pas file with the controller, the server and controller are guaranteed to have the same understanding of the various interfaces involved.*

is essentially a wrapper around a call to `IDispatch.Invoke`.

A Server's Eye View...

Right, so we've looked at life from the perspective of an OLE Automation controller, and it all looks very straightforward thanks to the magic being performed behind the scenes by the Delphi compiler and the runtime library. Now it's time to take a look at the state of play from the perspective of the MemoEdit application. How easy is it to implement an automation server using Delphi? Let's open the MemoEdit project and take a look.

The automation object which implements the application level (`IMemoApp`) interface is contained in the file MemoAuto.pas. As ever, bear in mind that an interface is not an implementation. The `MemoAuto` unit includes Memo_TLB.pas, the same type library definition as is included by the controller application. Thus, MemoAuto can 'see' the definition of `IMemoApp`, its job is to provide the implementation. With this in mind, look at the definition of `TmemoApp` in Listing 4.

This declaration states that `TMemoApp` is derived from `TAutoObject` and that it implements the `IMemoApp` interface. Because we're dealing with a Borland (as opposed to you-know-who!) development system, no surplus fat is in evidence. In other words, `IMemoApp` implements six custom methods and all we have to do is provide code for those six methods, we don't have to muck about providing declarations of low level `IDispatch` or `IUnknown` methods. All the grunt work is handled transparently by the `TAutoObject`. When implementing an OLE Automation server, `TAutoObject` is generally the best class to derive from. When you define methods of an automation object, all methods (apart from those inherited from `IDispatch` and `IUnknown`) must specify the `safecall` calling convention which automatically implements exception handling and errors in a manner compatible with OLE conventions.

As you can see, much of the code in `TMemoApp` is very straightforward. The six aforementioned methods just map down onto methods of the main form. Thus `Get_MemoCount` simply calls the main form's `MDIChildCount` method, `CascadeWindows` calls the main form's `Cascade` method, and so on. In fact, if you open the `MainFrm` unit, you'll find that, other than the comments, there's no hint that this is an OLE Automation server we're dealing with. And that's rather nice! By hiving off the automation object classes into other units, this small application represents an excellent example to follow when building your own Automation servers. You should avoid getting the OLE interface classes mixed up with the user interface code, keep them separate if you can.

More interesting are the three methods that return `OLEVariant` as the function result. `NewMemo` and `OpenMemo` both create new MDI child windows, while `Get_Memos` returns an instance of an existing child window. You can see that in all three cases, the method returns the `OleObject` property of the associated `TEditForm` window. In order to see what this corresponds to, we need to take a peek inside the `EditFrm` module where we find that the `OleObject` property of a `TEditForm` is defined like this:

```
property OleObject:
  Variant read GetOleObject;
```

This equates to the retrieval of a private field called `fMemoDoc` which is of type `TMemoDoc`. `TMemoDoc`, in turn, is an implementation of our old friend, `IMemoDoc`. A new `TMemoDoc` object is created by the Automation server every time that a native `TEditForm` window is created and it effectively acts as a 'proxy' for the real form when communicating with the Automation controller. The `TMemoDoc` object has a pointer to its owning form and `TEditForm` has a reference to its proxy. So, two-way mapping from one to the other is very simple.

The most important point in all this, of course, is that the Automation server and controller both include the same Memo_TLB.pas

➤ *Listing 4*

```
type
  TMemoApp = class(TAutoObject, IMemoApp)
  protected
    function Get_MemoCount: Integer; safecall;
    function Get_Memos(Index: Integer): OleVariant; safecall;
    function NewMemo: OleVariant; safecall;
    function OpenMemo(const FileName: WideString): OleVariant; safecall;
    procedure CascadeWindows; safecall;
    procedure TileWindows; safecall;
  end;
```

file and using this same type library information to create and access automation objects which are passed between them. If you've followed my tutorial on the undocumented `LibIntf` unit, or if you've done much work with Borland's Open Tools API, you'll realise there's an analogy here. The Delphi IDE and installed add-ins guarantee interoperability by including the same class declaration templates (such as those in `EditIntf`, `ToolIntf`, and so on) and in the same way Automation servers and controllers guarantee interoperability because they're both using the same type library.

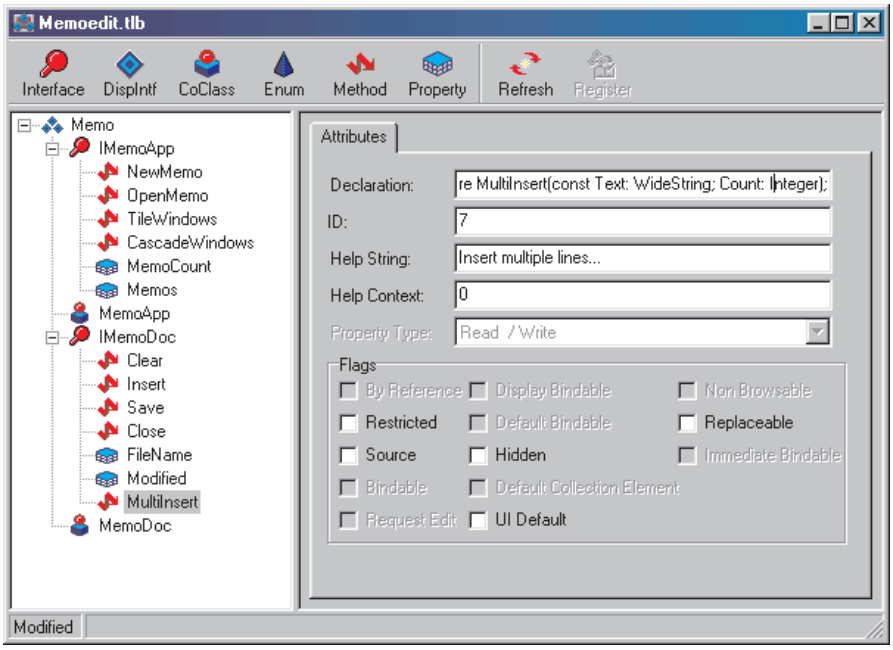
In Praise Of Type Libraries...

It goes without saying that, if the type library changes, then server and controller must both be updated. If you've still got the MemoEdit project open in the IDE, you'll see that there's a `Type Library` entry on the `View` menu: this gives access to Delphi's graphical type library editor. When working with the controller application, this menu option will be disabled which is just as it should be, the type library is owned by the server, not by the controller.

Let's dip our feet into the OLE Automation water by adding a new method to the `IMemoDoc` class. We'll introduce a new method, `MultiInsert`, to add the same line of text multiple times to a document window. It's defined like this:

```
procedure MultiInsert(
  const Text: WideString;
  Count: Integer);
```

Choose `Type Library` from the `View` menu to start the type library editor. If you haven't got a project currently loaded, you can open the existing type library by simply doing a `File|Open`, selecting `Type Library` in the drop-down file type list and choosing the existing MEMOEDIT.TLB file. Make sure the `IMemoDoc` interface is selected and then click the `New Method` button at the top of the window. This will add a new method to the interface and give it a default name. Type the above line into the `Declaration` box, give it a dispatch ID of 7 and enter an optional help string. That's it! You've just added a new method to an interface. It wasn't so hard, was it?



➤ *Figure 3: Here you can see the addition of the MultiInsert method to the existing type library. As you work with the type library editor, you can press the Refresh button to see the changes immediately reflected in any open window that contains the Pascal implementation unit for the type library. The rather cute graduated caption bar is down to Windows 98, by the way!*

If the Memo_TLB.Pas file is loaded, you can click the `Refresh` button to see Delphi automatically update the implementation representation of the type library. Click `File|Save` when you've finished with the type library editor. This will regenerate the Pascal implementation file and update the binary .TLB information at the same time.

Of course, if we try building the server now, the compiler will complain that no `MultiInsert` method has been defined in the `TMemoDoc` class, so add it now. Be sure to include the `safecall` specifier at the end of the declaration. If you forget, the compiler will complain that the declaration differs from that in `IMemoDoc`. Listing 5 shows the declaration for `MultiInsert`: not exactly rocket-science, but bear in mind that at this point I'm simply trying to familiarise you with the basics of OLE Automation.

With this simple change, the server will compile and run. Now open the controller application and try modifying the `TMainForm.AddTextButtonClick` method so that it looks like Listing 6.

You can now compile and run the controller. As you'd expect, you'll find that every time you click the `Add Text to Memos` button, five copies of *'Hello Automated World'* will appear in each memo window.

Conclusions

In this month's instalment, I have described the basics of OLE Automation and shown how late binding and early binding both have their part to play in making things happen.

Even if your primary interest is in developing ActiveX components, it's still important to have a solid grasp of OLE Automation technology. The reason for this is simple: OCX components are effectively pocket-sized Automation servers which (because an OCX file is just a renamed DLL) function as in-process servers within the containing application. Container applications interact with OCX components using the same OLE technology that we've been looking at.

```
procedure TmemoDoc.MultiInsert (const Text: WideString; Count: Integer);
begin
  while Count > 0 do begin
    Insert (Text);
    Dec (Count);
  end;
end;
```

➤ *Listing 5*

```
procedure TMainForm.AddTextButtonClick(Sender: TObject);
var I: Integer;
begin
  for I := 1 to 3 do
    if not VarIsEmpty(Memos[I]) then
      Memos[I].MultiInsert('Hello Automated World' + #13#10, 5);
end;
```

➤ *Listing 6*

As you'll no doubt realise at this point, OLE Automation might have been an interesting alternative for building add-ons for the Delphi IDE. Rather than using the current approach, imagine a world where the IDE treats each add-in as a loadable OCX component. Interestingly, there are some indications in Delphi 3 that Borland may be moving in this direction. For example, the main IDE window is based around a class which is called `TAppBuilder`. This class has an automation table (part of the runtime type information associated with Delphi executables) which contains the following two entries:

```
function DesignForms(Index:
  Integer): OLEVariant; safecall;
function DesignFormCount:
  Integer; safecall;
```

As you can see from the function names, this is an interface designed to provide access to the currently loaded design forms within the IDE. It's questionable whether this is a hint of what we can expect in Delphi 4, or whether it's merely a vestige of some failed experiment that never came to fruition. Time will tell! Next time round, we'll look at some more sophisticated examples of OLE Automation, possibly involving unspeakably naughty happenings in the Delphi IDE.

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* which is also published by iTec. You can contact Dave at Dave@HexManiac.com