The Delphi 3 Novelty Store: 2

by Brian Long

Last month we saw the splendid new environment in Delphi 3 and spent some time looking at its new Code Insight features and exploring the VCL's new Business Insight architecture. This month it is time for Active Insight.

Before we start I must mention again that this article is based on pre-release software and so some parts might be inaccurate when compared with the shipping product (although there are no firm details at the time of writing, we hope that Delphi 3 may be released by the time you read this). There are some notes at the end of this article on some aspects of last month's installment which need updating.

Are you sitting comfortably? Then I'll begin.

Active Insight

This seems to be the all-new Borland marketing term for the Internet, COM, DCOM, ActiveX and distributed application features in Delphi 3. A lot of the COM and DCOM support needs a good knowledge of COM and OLE mechanics to appreciate. Or perhaps it just needs someone with a good knowledge of COM and OLE to explain it well... Anyway, I will mention the salient factors and leave it to someone else to explain all the gory details with useful examples at length in a future article. I cannot pretend to have a full understanding of all these features yet, but I will strive to accurately represent what is available in the new version.

COM, DCOM And OLE

The Component Object Model (COM) is based around a binary standard and is intended to provide a way of making classes available to executables other than the one they are defined in. Being a binary standard and not a source code standard means that classes can cross language barriers and hopefully platform barriers.

The binary standard dictates how a class virtual method table (VMT or v-table) is laid out and also dictates how COM objects communicate. Additionally, to be COM-compliant a class must have a registered class ID, be able to be externally instantiated by a class factory and support lifetime management.

DCOM, or Distributed COM, allows a COM class to be instantiated on a different machine. OLE (Object Linking and Embedding) is a specific implementation of COM that Microsoft have created, giving many facilities over and above what COM requires.

Interface Support

In order to properly support COM and DCOM, Borland have now specifically implemented the concept of a class interface. If you have played with the Tools API you will be familiar with the notion of a class interface: a class that can define public methods and properties, but with no method implementations or data fields. The idea is to represent the interface to a class with no hint as to how or where it is implemented. The implementation is done elsewhere.

It is quite a similar concept to an import declaration for a routine in a DLL. In that case you specify the interface to a subroutine, however you typically also specify where the routine is implemented. With an interface you do not need to know, let alone specify, where the class is implemented. That information is stored in the Windows registration database (if the class is a COM class).

To support interface classes better, Borland have added a new data type: the interface. An interface is much like a class, but is defined with the already reserved word `interface` instead of `class`. The naming convention is that classes start with a T (for Type) and interfaces start with an I (for Interface). The rules for what can go in an interface are much as just described above, although you can optionally specify an interface ID (IID).

An IID is one of two commonly used types of GUID and uniquely identifies an interface class. All the registered IIDs can be found in the Windows registry under `HKEY_CLASSES_ROOT\Interface`. A GUID is a globally unique identifier that is stored in the registry and used to identify interfaces and COM classes. GUIDs are 16 byte binary values and are specific instances of universally unique identifiers (UUIDs). The `System` unit defines a `TGUID` record type (see Listing 1) although GUIDs are typically represented in a string format, for example:

```
'{2835826F-7E60-11D0-9FEA-A42B00C10000}'
```

Delphi 3 has the functions `GuidToString` and `StringToGuid` in the `ComObj` unit to convert between these formats (the `OLEAuto` unit has the equivalent `ClassIDToString` and `StringToClassID`). As a convenience measure a `TGUID` typed constant can now be initialised with an appropriately formatted string. This means that instead of writing this rather cumbersome expression:

```
const
  AGuid: TGUID =
 (D1:$2835826F;D2:$7E60;
  D3:$11D0;D4:($9F,$EA,$A4,
  $2B,$0,$C1,$0,$0));
```

you can write the more naturally GUID-like:

➤ *Listing 1*

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

```
const
  AGuid: TGUID =
 '{2835826F-7E60-11D0-9FEA-A42B00C10000}';
```

Additionally, when calling a procedure or function that takes a constant or value parameter of type `TGUID`, an interface type (such as `IUnknown`) can be passed and the compiler will extract the IID and pass that instead.

The other type of GUID is a class identifier or class ID (CLSID) and is stored in the registry by an OLE server to identify an application that implements a COM class. These can be found under `HKEY_CLASSES_ROOT\CLSID`.

The pre-defined Windows COM interfaces were declared in the Delphi `OLE2` unit as normal classes (there was no previous alternative). Now the new `ActiveX` unit and `System` unit re-implement them as proper interfaces.

In much the same way as all classes inherit ultimately from `TObject`, all interfaces inherit ultimately from `IUnknown`, the most basic interface. Listing 2 shows the old `OLE2` and new `System` unit versions of `IUnknown`.

Note that in an interface you can specify an `out` parameter where we might normally use a `var` parameter. `Out` parameters are slightly less efficient than `var` parameters because the compiler generates code to automatically resource manage the `out` parameter reference. Because of this it is recommended you only use `out` parameters with COM object methods.

Interfaces can inherit from one another in just the same syntactic way as classes, but there is no concept of increasing the functionality of an interface, because it has no functionality. An interface that inherits from another interface just makes a bigger interface.

Multiple Inheritance?

A question arises as to how you actually use interfaces. Interface definitions exist to represent the well-defined interface to some object. This object may exist in a DLL, in another EXE or indeed on another machine. Alternatively you may need to implement the interface in a real class yourself in your own EXE. To do this requires some syntax that looks very much like multiple inheritance (uh-oh) as shown in Listing 3. In fact Delphi 3 does not support multiple inheritance, but it does allow you to specify that a class will implement one or more interfaces by specifying those interfaces along with the ancestor class in the class declaration. You still cannot inherit from more than one class (phew!).

If one interface is included several times into a class due to it being specified in some of a class's ancestors, the compiler ensures that it is only considered once. Once you have specified that a class implements an interface, you need to implement the methods that it defines. Unfortunately, because you can specify multiple interfaces you may get a clash of methods from different interfaces with the same name.

In order to resolve this issue you can employ method resolution clauses. This allows you to specify which class method will implement which of the clashing interface methods. Listing 3 shows the idea. These can also be used to simply implement an interface method using a class method of a different name.

Lifetime Management

Since all interfaces inherit ultimately from `IUnknown`, any class that implements an interface will have to implement the methods shown in Listing 2. `_AddRef` and `_Release` should respectively increment and decrement an internal counter. This allows the implementation of an interface to keep track of how many references exist to an interface. When the counter reaches zero, the object should free itself.

As another convenience measure the `System` unit defines a class `TInterfacedObject` which inherits from `TObject` and implements `IUnknown`'s methods (there is an

➤ *Listing 2*

```
//As defined in the OLE2 unit
IUnknown = class
public
  function QueryInterface(const iid: TIID; var obj): HResult; virtual;
stdcall; abstract;
  function AddRef: Longint; virtual; stdcall; abstract;
  function Release: Longint; virtual; stdcall; abstract;
end;

//As defined in the Delphi 3 System unit
IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}'] //This is IUnknown's IID
  function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

➤ *Listing 3*

```
IInterface1 = interface
  procedure NotAClash;
  procedure NameClash;
end;
IInterface2 = interface
  procedure UniqueName;
  procedure NameClash;
end;
TExampleClass = class(TComponent, IInterface1, IInterface2)
  procedure IInterface1.NameClash = NameClashA;
  procedure IInterface2.NameClash = NameClashB;
public
  procedure UniqueName;
  procedure NotAClash;
  procedure NameClashA;
  procedure NameClashB;
end;
```

associated class reference TInterfacedClass).

The _AddRef and _Release methods in TInterfacedObject implement the required reference counting whilst QueryInterface checks to see if a specified interface is implemented by the class and if so returns a reference to it. So this class is a good base class for COM-style objects. TInterfacedObject itself is not a COM class since it does not have any class factory support.

Much like the mechanism for defining object references you are also able to define interface references. You can assign one interface reference to another and Delphi automatically generates code to call _Release for the old interface being overwritten (if there is one) and _AddRef for the one being assigned. Additionally, if an interface reference goes out of scope, _Release gets called. This means that you can forget about having to destroy objects if you talk to them solely through interfaces. Additionally, you can forget about _AddRef and _Release because they get called automatically for you.

Given an object that implements a certain interface, you can assign the object reference to an interface reference provided the declared type (not the actual type) of the object reference implements the interface. This is checked at compile time and an error is generated if the requirements are not met.

Interface Querying

The as operator normally gets used in an expression like ObjectRef as ClassType. Listing 4 shows an example statement and its effective implementation.

Delphi 3 gives us another use for the as operator. You can get hold of an interface reference from an object reference or another interface reference using as. This is called interface querying and allows a more flexible way of getting interfaces than the approach described above. It caters for object references that are declared of an ancestor type and would give compile time errors using direct assignment. The expression:

```
ObjOrIntRef as InterfaceType
```

will attempt to resolve to an interface reference of the specified interface type, or nil, so long as one of the following are true:

- ObjOrIntRef is a reference declared as a class type that implements IUnknown, or
- ObjOrIntRef is a reference declared as an interface type.

In order for the compiler to ensure these requirements are met the expression gets compiled as:

```
IUnknown(ObjOrIntRef) as
  InterfaceType
```

If the reference is nil, the result of the whole expression becomes nil. If not, a call is made to the reference's QueryInterface method which gets passed the IID of the specified interface. If the interface is found, the assignment is successful, otherwise an EIntfCastError exception is raised. Listing 5 shows an example statement and its corresponding implementation.

OLE Automation

The OLEAuto unit is still supplied in the VCL for Delphi 2 compatibility, but now it is in Delphi 3's LIB\DELPHI2 directory which must be added to the unit search path. However, OLE automation is now better achieved using the ComObj unit. For a start, the OLEAuto unit cannot be compiled into a package and so therefore neither can anything that uses it. ComObj offers a CreateOLEObject function that can be used just as it could before, however the new one returns an IDispatch interface as opposed to a variant. This IDispatch can still be assigned to a variant (after all, the variant returned by Delphi 2's CreateOLEObject contains an IDispatch reference).

An IDispatch interface represents an object that supports OLE automation, ie a COM object that has an Invoke method which can dispatch calls to appropriate routines at runtime.

Delphi 3 does things slightly differently to the earlier version when creating an OLE server (via the Object Repository which looks like Figure 1). Rather than just manufacturing one unit with a TAutoObject descendent in, it now makes two units. The new interface library unit contains an interface inherited from IDispatch that describes your OLE Automation server COM object. It also implements a simple coclass (component ob-

Listing 4: The as operator and its implementation in normal use

```
var
  List: TListBox;
  Sender: TObject;
...
List := Sender as TListBox;   // this is implemented by the following code
if Sender = nil then
  List := nil
else
  if not Sender.InheritsFrom(TListBox) then
    raise EInvalidCast.Create('Invalid class typecast')
  else
    List := TListBox(Sender);
```

Listing 5: The as operator in its new guise of interface querying

```
var
  Dispatch: IDispatch;
  OLEServer: TAutoObject;
...
Dispatch := OLEServer as IDispatch;  // this is implemented by the following code
const
  IDispatchID: TGUID = '{00020400-0000-0000-C000-000000000046}';
...
if OLEServer = nil then
  Dispatch := nil
else
  if IUnknown(OLEServer).QueryInterface(IDispatch, Dispatch) <> 0 then
    raise EIntfCastError.Create('Interface not supported')
```

ject class) that uses OLE system calls to create an instance of your server object on the current machine (Create, for COM support) or on a remote machine (CreateRemote, for DCOM support).

This library unit can be used in any other Delphi project that needs to talk to the object. The unit is generated as a translation of the project's type library (binary .TLB file) which is maintained by Delphi, and gets regenerated whenever the type library is refreshed. Because of this, you should not add anything into the unit as it will be overwritten.

The other unit is much like it was in Delphi 2 except the TAutoObject descendant also implements your new COM interface. Also the initialisation section creates a class factory specific to your OLE server.

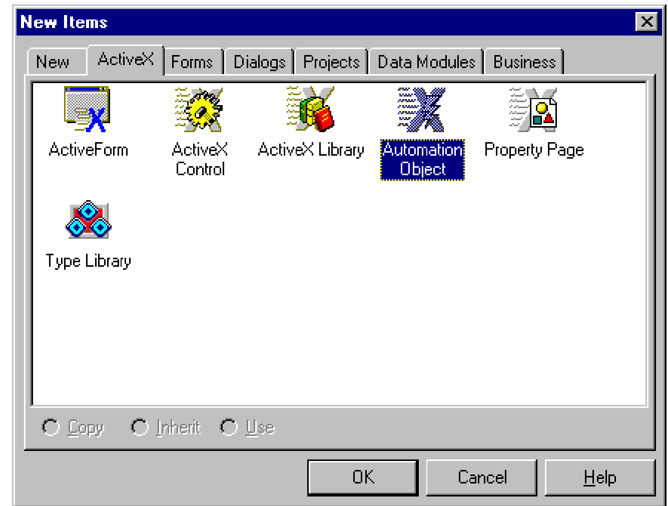
In order to add new properties and methods into the interface and class simultaneously, you can get Delphi to do all the typing with Edit | Add To Interface... or by right clicking and choosing Add to Interface... when in the class implementation unit. You can also use the type library editor (see later).

When your server is complete it needs to have its CLSID, IID and also its type library ID and some other miscellaneous bits of data stored in the registry. Just running the server will do this, but it is usually better to run the server with a /regserver parameter. This ensures that the server starts, registers and terminates without messing around creating forms and thereby potentially taking a long time.

Given a project XXXX.DPR which implements an interface IYYYY in a COM class TYYYY, the CLSID is defined in the library unit as the constant Class_YYYY. The IID is specified in the interface: you can just refer to IYYYY if you need access to the IID. The type library ID is a constant called LIBID_XXXX. The interface unit itself is called XXXX_TLB.PAS and is generated from the type library XXXX.TLB.

In order to create COM objects on remote machines you need some DCOM plumbing to be present. Windows NT 4 has this out of the box but Windows 95 does not. You will need to use Borland's OLEnterprise package or get hold of Microsoft's DCOM for Windows 95 package.

➤ Figure 1




```
uses
  ComObj; //used to use OLEAuto
var
  //This is a variant and uses the IDispatch.Invoke method
  //When OLE properties/methods are accessed, they are packaged
  //into calls to VarDispInvoke in OLEAuto or ComObj
  AutoOleServer: Variant;
...
AutoOleServer := CreateOleObject('Auto.Server');
```

➤ Listing 6

```
uses
  ComObj,
  Auto_TLB; { Auto-generated server object interface definition unit }
var
  //Note the variable is an interface inherited from IDispatch
  AutoComServer: IServer;
...
//This internally uses QueryInterface to get an IDispatch descendent
AutoComServer := CreateOleObject('Auto.Server') as IServer;
```

➤ Listing 7

Dual Interfaces

Historically, Delphi's only solution to controlling an automation server was to use CreateOLEObject in association with a variant containing an IDispatch. All the calls to methods and properties you write in your source code are packaged up into call descriptors with associated parameter lists. The compiler generates code to call the System unit routine _DispInvoke which simply calls whatever routine the VarDispProc pointer points to, giving it the relevant call information.

VarDispProc defaults to pointing at some code that generates runtime error 222 or, more usually, an EVariantError exception. When either the OLEAuto or ComObj unit is added to a uses clause in your project, their initialization sections ensure VarDispProc is routed to a procedure they both implement called VarDispInvoke.

This uses the IDispatch.Invoke method to attempt to call the relevant routine and pass the appropriate parameters. If it fails, you get a run-time exception. The point being that the appropriate routines are only located and verified at runtime. The compiler is unable to check the validity of calls or parameters as it has no declaration available for the automation server object. This is often referred to as

late binding, or run-time call resolution.

Now that we have interfaces available, we have another option. Provided we can get hold of an interface for the COM object then we can take advantage of early binding, or compile-time call resolution. The compiler will generate code to directly access the methods via the virtual method table.

When you call ComObj's CreateOLEObject you can assign the return value directly to an interface reference variable (using the as operator as described earlier to perform interface querying). Then when you call a method or property of the object the compiler can verify that it actually exists in the interface and call it directly.

A COM object is said to be accessible through dual interfaces if you can access its methods and properties through both the IDispatch.Invoke method (usually via a variant) and also the VMT (via an interface definition). Listings 6 and 7 briefly show some syntax for creating an instance of an OLE server using the two approaches, where the class was specified simply as Server when created in a project AUTO.DPR.

If you do have an interface for the target COM object, which may or may not support OLE automation (ie may or may not implement the IDispatch interface) then you can also opt for CreateComObject or CreateRemoteComObject. These take class IDs, not ProgIDs (remember these are available as constants in the interface unit).

Because Delphi also makes a simple component object class in the interface library unit, there is one extra alternative which amounts to the same as Listing 7. The call to CreateOleObject in the listing can be replaced with:

```
AutoComServer :=
  CoServer.Create;
  { or CreateRemote(
    'Remote Machine Name') }
```

The safecall calling convention is used to implement methods of dual interfaces in Delphi. This is automatically dealt with when using the Delphi OLE Automation tools. Safecall is like a cross between register and stdcall. It operates just the same as stdcall but uses CPU registers in preference to the stack for passing parameters. Additionally, a safecall function will have a default return value (unlike other functions) of S_OK (which is defined as 0) and can deal with errors in an OLE-safe way.

If you did not write the COM object or OLE server that you need to talk to then you can generate an interface for it so long as you have a type library (see later) for it.

COM Classes

TComObject implements the basic functionality required for COM objects. This includes being registered and the ability to be externally created by a class factory (there is a TComObjectFactory class available for this). It also supports OLE exception handling, aggregation (the Controller property returns an IUnknown reference to the controller object) and the safecall calling convention necessary for dual interfaces.

TTypedComObject inherits from TComObject and implements the IProvideClassInfo interface to provide type information. A type library is necessary for this class. Delphi supports making type libraries as explained later.

The variable ComServer is an object of type TComServer and is defined and created in the ComServ RTL unit. The ComServ unit is used in COM servers (modules that implement COM or OLE classes) and can only be used in an EXE or DLL: it cannot be compiled into a package. ComServer replaces the Automation object from the OLEAuto unit and can be used to identify why the server was launched (for example for the purpose of OLE Automation or class ID registration). Its main job is to track which COM or OLE objects get instantiated and destroyed and generally make your project act like a COM server should.

Class Factories

Class factories are used to create instances of a specific COM object type. A class factory is restricted to creating instances of a specified class or descendants of that class. The base class is specified in the constructor for the class factory. When another application calls the Windows CoCreateInstance or CoGetClassObject APIs, or the equivalent Delphi CreateComObject or CreateOleObject functions a class factory is used to create the object in the server app.

Delphi supplies a class factory type for the COM class types it implements. The class factories in a Delphi application are held in a list in an object of type TComClassManager called ComClassManager (from the ComObj unit). You can use ComClassManager to locate the class factory for a given class type or class ID. The ComServer object performs necessary lifetime management on the factories in ComClassManager's list. For example when the last COM object in the server is destroyed, ComServer frees all the factories and terminates the program.

The automation class that gets auto-generated in a Delphi 3 OLE Automation server has a class factory manufactured for it. The initialisation section of one of the units calls:

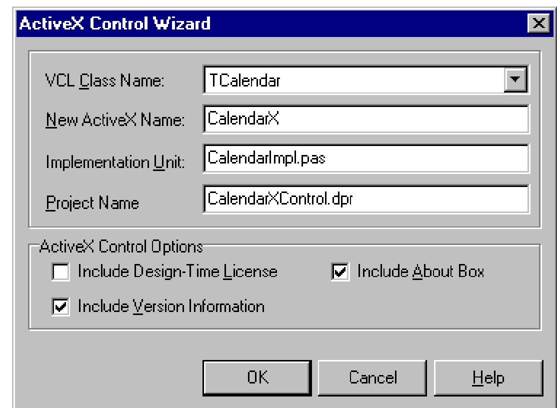
```
TAutoObjectFactory.Create(
  ComServer, TServer,
  Class_Server,
  ciMultiInstance)
```

This creates a TAutoObjectFactory for the TAutoObject descendant type TServer where Class_Server is the OLE Automation class ID and the last parameter dictates how many instances of type TServer can be manufactured by one server application.

Because of class factories, we have several options for instantiating a COM object. We can simply call the constructor for the class (this must be done if the class does not have a registered class ID). The constructor for TComObject actually uses the relevant class factory to construct itself. We can manually use the class factory to construct an instance. Alternatively we can call CreateComObject or CreateOleObject from the ComObj unit,

which boil down to Windows OLE DLL calls. Windows will indirectly use the class factory to create the object as well. Each of the statements in Listing 8 would be acceptable to create an instance of a `TServer` class that implements the `IServer` interface which has a registered class ID `Class_Server`.

The first two options are only viable if the class is implemented in the current EXE or DLL that we are writing.



➤ *Figure 2*

```
var Server: IServer;
...
//Construct the object directly
Server := TServer.Create;
Server :=
  TServer.CreateFromFactory(ComClassManager.GetFactoryFromClass(TServer),
    nil);
//Use the class factory to create the object
Server := ComClassManager.GetFactoryFromClass(TServer).CreateComObject(nil)
  as IServer;
Server :=
  ComClassManager.GetFactoryFromClassID(Class_Server).CreateComObject(nil)
    as IServer;
//Use Windows to create the COM object
Server := CreateComObject(Class_Server) as IServer;
```

➤ *Listing 8*

One Step ActiveX

An ActiveX is the new term used for what used to be called an OCX (with a few differences here and there). This means that ActiveXs can be embedded into Visual Basic and C++ applications (amongst others), as well as Web pages. OCXs and ActiveXs are both special cases of in-process OLE servers: OLE objects in DLLs that act as visual controls. Historically, OCX and ActiveX controls have been tricky to write. The development kits were written for C++ developers and support in Delphi was left to third parties. Now ActiveX controls are as easy to make as native Delphi components.

To make an ActiveX control you need an ActiveX Library open (a project set up to make an ActiveX binary DLL file). You can choose one from the ActiveX page of the Object Repository. If you do not have one set up, one is made for you when you choose ActiveX Control from the repository.

Having chosen to make an ActiveX, the ActiveX Control Wizard (Figure 2) asks you for the VCL component class to base the ActiveX on, the name of the ActiveX, the unit name and project name (if there is no ActiveX Library open). You can also request an About box form to be added in along with version information and design-time licensing support. The list of VCL classes will include only `TWinControl` based components that do not rely on being connected to another control.

When the ActiveX files are generated, the code in the various units is reasonably sizeable, with much of it being dedicated to surfacing the component properties into the ActiveX properties. Once the ActiveX is complete and you have added all the code that you want to, you can use `Register ActiveX Server` from the `Run` menu to make the control available to other applications. `Unregister ActiveX Server` removes any trace of the control from the registry.

Additionally, the `Project` menu has `Web Deploy Options` and `Web Deploy` menus. The `Web Deploy Options` item lets you specify a target directory for the OCX and one for a sample HTML page that links to the ActiveX. You can also specify a URL that will be used when the HTML page is accessed via the Web. Additionally you can choose to employ Microsoft Cabinet compression and code signing (which requires a credentials file) and specify which, if any, additional files (such as packages) to deploy. The `Web Deploy` menu item deploys your files according to the chosen options.

Property Pages

You can go back to the Object Repository and choose to make a property page for your ActiveX. This is a form-like thing that can be used in the design-time environment of whatever product is making use of the ActiveX, rather like a component editor, typically to set the values of several properties. A `TPropertyPage` descendant has two methods for taking changed property values from the user (`UpdateObject`) and for surfacing property values into the property page (`UpdatePropertyPage`).

The property page is invoked from the ActiveX control's `DefinePropertyPages` method. Comments in that method implementation tell you that you simply need to use the property page unit and call `DefinePropertyPage`, passing the class ID constant of the property page in question.

ActiveForm

ActiveX controls allow you to package a VCL component into an embeddable control. We also have the option of making an ActiveX that contains a whole form, called an ActiveForm. This again comes from the ActiveX page of the Object Repository and offers you similar

choices about target file name, and design-time license support etc.

There are some differences in properties between a form and an ActiveForm. For example, `BorderStyle` is replaced with `AxBorderStyle` (which has different values: `afbNone`, `afbSingle`, `afbRaised` and `afbSunken`). Also `ClientHeight` and `ClientWidth` are gone as are `FormStyle` and `WindowState`.

The current joke is that if you don't know Java and aren't too creative with HTML or JavaScript, you can write all your fancy Web stuff in a Delphi ActiveForm and get your Web page to link to the resultant ActiveX control.

For an Intranet this would probably be a reasonable approach. If other Delphi applications abound in the company, then ActiveForms and the package support should make any required downloads from the network or Internet reasonably small: the package files are only required once on any machine. Things are potentially even better if you put the packages on an available network drive.

Type Libraries

OLE supports the concept of a type library. This is a binary file that allows other development tools to easily create interfaces to your classes and allows applications to find out what interfaces, methods and properties a COM class supports. Standard binary type libraries have a .TLB extension and can be shipped as separate files, or linked into applications as resources. Delphi automatically creates type libraries for OLE Automation servers, ActiveX controls and ActiveForms and binds them to the EXE with a `$R` directive. You can also create a type library from the Object Repository or open an existing type library using `File | Open...`

The type library editor allows you to make and customise interfaces, COM classes, dispatch interfaces and other attributes and entities that can go in a type library as shown in Figure 3. The information stored in the library is used to manufacture the Pascal library interface unit mentioned before.

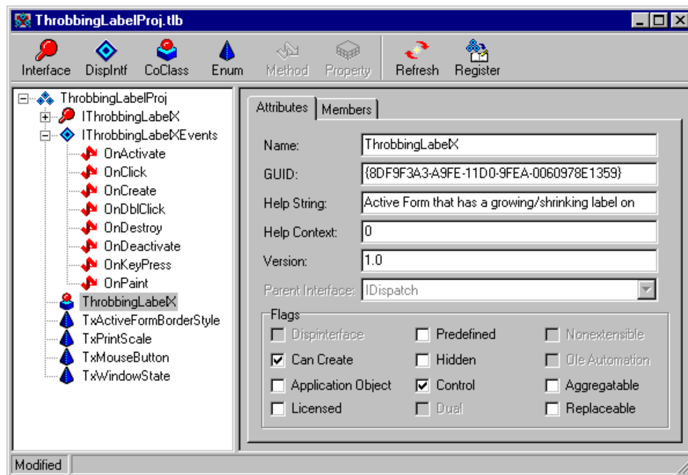


Figure 3

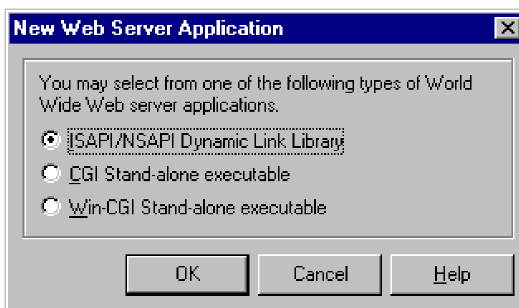


Figure 4

For a project called ComApp.DPR, the library unit will be called ComApp_TLB.PAS. As you make modifications to the type library, you can push the `Refresh` button to update the library unit (along with any other dependent units, such as remote data module units or automation server object units).

In much the same way as `F12` toggles between a form and the corresponding form unit, `F12` also toggles between the type library editor and the library interface unit.

To generate Delphi interface definitions for third-party COM objects, simply open up the type library in Delphi and press `F12`. Delphi allows you to open type libraries stored in .TLB and .OLB files and also compiled into .DLL, .OCX and .EXE files. Additionally, there is a command-line tool TLIBIMP.EXE supplied in Delphi's BIN directory that does a similar translation. As another possibility, you can choose `Project | Import Type Library...`, which lists all the registered type libraries (although you can still choose others) and will (by default) place the appropriate interface unit into the Delphi's IMPORTS directory.

Web Server Applications

I cannot do these justice in the available space but they will no doubt be covered later by others.

The idea is to enable simple generation of a server-based add-on for your Web server. The Web server application can generate new HTML pages in any way it sees fit, but often by talking to database tables for some of the content. The support components in Delphi 3 make it easy for the server application to maintain many simultaneous database connections.

If you do not have a Web server, then you can download the free single-user Microsoft Personal Web Server (size 756Kb) from:

```
http://www.microsoft.com/msdownload/
  ieplatform/iewin95.htm
```

To make a Web server app, choose `Web Server Application` from the Object Repository and you are then faced with a dialog (Figure 4) that gives you the option of making a DLL that supports both Microsoft's ISAPI and NetScape's NSAPI APIs. You can also choose to make a CGI or WinCGI application. All options present you with a Web Module in the project, something

that looks rather like a Data Module. The Web module has an inbuilt TWebDispatcher component, which can be manually added to other forms or data modules. In order to build up the logic in the application you can use the Web Module's Actions property (in reality the WebDispatcher's Actions property) to build up possible elements that may be passed in the command-line, or path string. Each action has an OnAction event handler that will be triggered to process it.

The Web Module has a couple of events of its own, but the rest of the functionality is implemented with some new high level Internet components: TPageProducer, TQueryTableProducer and TDatasetTableProducer. These can be used to generate HTML to produce nicely formatted Web pages as required, potentially using information from various data sets to make data-aware Web pages.

TPageProducer is a generalised HTML manufacturer where you can have a fixed block of HTML in the HTMLDoc TStrings property. This can include HTML custom tags which can then be dealt with individually via the OnHTMLTag event to customise the generated HTML. The TQueryTableProducer component is connected to a query and will automatically generate a nicely laid out HTML table showing the current result set (or at least the values for the fields specified for inclusion in the table) when referred to.

To make a server app for all flavours of server API simply requires you to make one project for each format and then add in a finished Web Module to each one. The underlying VCL architecture takes care of the rest.

Update On Part 1

Last month's initial foray into Delphi 3 was written using a field test several versions away from completion. Consequently, some things have changed. Here is what I have found to be different so far.

The .DFR and .STR files that were mentioned as being temporary in nature but were left in your project directory are now deleted by Delphi, so you can forget about them. However, the online help for the current field test I am using suggests that to help string translation resource strings might get stored in a .DRC file.

The suggestion for getting the resource id of a resourcestring does not seem to be as simple as suggested. Incorrect values are returned, so I'll leave that on the back burner for now.

The Package Collection Editor (PCE) was described as being able to collect several .DPL files into one big binary file. This is true, but it is in fact more useful. Since a .DPL on its own is only useful to a compiled EXE (the developer needs .DCU and .DCP files) the PCE allows you to also include a group of files related to the .DPL, which might typically include some of the following: .DPK, .PAS, .DCU, .DFM and .DCP files.

For distributed datasets, because things were still changing, the details of how to set up the server and client were rather vague to say the least. In the server application you need to add in a Remote Data Module (as distinct from a Data Module). A remote data module is a COM class that implements an interface designed to surface your TProvider object. When it asks you for a class, I will assume you choose RDM and your project is called DataServer. This makes a remote data module called RDM of type TRDM with an interface called IRDM. Having added a TQuery and a TProvider, you connect the provider to the query with the provider's DataSet property. To make the provider available through the interface, right-click on the query and choose Export query from data module. This updates the type library and interface unit and then implements the appropriate method in the data module class for you.

If you don't like the name of the exported property (it will be the same as the name of the query) then load up the type library (View | Type Library), expand the appropriate interface and modify the property declaration. When you are happy, press Refresh and all the code will update. This property name is needed in the client application.

I mentioned last month that the client's RemoteServer component has a ServerName property that needs to be set to the ProgID of the server. This will be DataServer.RDM. If the server app is registered, the ServerName property editor will find all the appropriate ProgIDs and list them out for you. Once chosen, it then fills in the ServerGUID property for you. The TClientDataSet connects to the TRemoteServer via its RemoteServer property: this basically links the server and client application together. To choose the appropriate provider, you set the ClientDataSet's ProviderName property name to match the remote data module interface property we were playing with just above. That sets the whole thing in motion: set Active to True and the data is read across the COM link.

Open Environment

At the high end of the Delphi 3 product line, Borland will supply some of the software they inherited from recently acquired company Open Environment.

The Entera product suite allows you to build distributed applications with the various component parts residing on a variety of platforms, communicating by a variety of standards. It supports DCOM, CORBA and DCE. It enhances normal DCOM to add in a naming service like the one which CORBA provides. The beauty of Entera is that the developer and the user can ignore the complexities of talking across these various protocols and platforms and it allows you to break away from the Microsoft platforms.

OLE Enterprise (which is written OLEnterprise) is a part of Entera that will be shipped with at least one level of Delphi 3: this can be used as an alternative for DCOM and has one or two useful things in its favour. Firstly, though the DCOM support in NT is adequate,

that for Windows 95 is, well, tricky. Many people have problems getting it to work sufficiently (or at all, as I can attest). Secondly, DCOM requires the application to specify the machine on which the COM object can be located. So you need one call (`CreateComObject`) for COM and another (`CreateRemoteComObject`) for DCOM.

OLEnterprise manages to allow the client program to use the same code for local COM communication as for remote COM. In other words, `CreateComObject` can be used regardless of where the target COM object resides. OLEnterprise sets things up in the registry to enable it to get the communication between machines up and running as and when necessary. Very briefly, this is how OLEnterprise is used to get a client app on one machine talking to a server app residing and registered on another machine.

With OLEnterprise installed on both machines, launch the Object Explorer on each. On the server machine, locate the server object in the Object Explorer, right click it and choose `Export` (this updates its registry entry). On the client machine's Object Explorer, choose `Registry | Connect...` and specify the server machine. This should allow you to see the server machine's exported registry entries. Select the appropriate entry, right-click it and choose `Import`. This adds appropriate registry entries on the client machine. That's all we need the Object Explorer for so they can now be closed.

During the OLEnterprise set-up, you are told that the Object Factory can be set up to launch as Windows starts: this should be done on the server. When the client application attempts to communicate with the server object, the registry entries that are encountered actually start it talking to a local OLEnterprise DLL called the Object Agent. This Agent then talks to the appropriate server machine's Object Factory. The factory creates the COM object and communication can then proceed via the Agent/Factory link with the client application being none the wiser.

One additional program that comes with OLEnterprise is called the Business Object Broker. This acts as a directory of objects that can provide location transparency and server failover. What this means is that you can have many copies of server applications distributed around an enterprise and the broker will direct the client to an appropriate one, and sort things out when the server breaks.

Conclusion

There is a lot of new stuff in Delphi 3 and it will take a long while to digest it all, but hopefully some of the details in this and last month's articles will help you on your way.

Brian Long is a UK-based freelance Delphi and C++ Builder consultant and trainer. He is available for bookings and can be contacted by email at blong@compuserve.com

Copyright ©1997 Brian Long
All rights reserved