The Delphi 3 Novelty Store: 1

by Brian Long

Delphi 3 is Borland's new incarnation of the award-winning RAD tool we all know and love. It has been quite well known that Borland C++ Builder and Delphi 3 have both been developed over the same period of time for delivery at similar times. It has also been fairly well known that the code names for these two products were Ebony and Ivory respectively. Today we are going on an Ivory hunt.

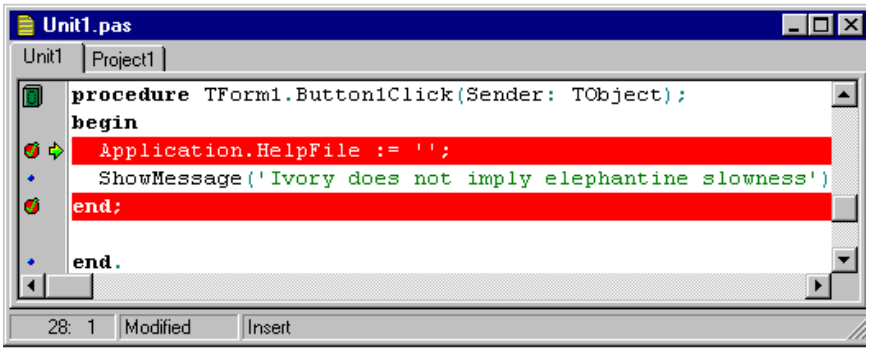
Borland have put an awful lot of effort into adding new features and functionality to what was already in place in Delphi 2. The general areas include EXE generation, database and Internet-related things. There are so many new bits and pieces that I cannot fit them all into one article, so some information will have to wait until next month.

These previews are written on the basis of pre-release product versions and, amongst others, the new database features are quite, how can I put this delicately, *dynamic* in their nature. As a consequence it may turn out that some bits and pieces are a bit different in the shipping product to the descriptions in these articles. I understand that most of the pending changes are specifically aimed at simplifying the use of various new product features.

First this month, we will take a look at the new environment.

IDE Updates

The Delphi development environment has had a face lift. The first thing you notice when you launch the IDE is that all the speed buttons on the speed bar and components on the component palette (which are implemented as speed buttons) have the new flat look that Microsoft has added to Internet Explorer 3.0 (see the screen shot on this month's cover). The `TSpeedButton` component has a new property called `Flat` to allow you to take on this look and feel. The `TDBNavigator` also has a `Flat` property



➤ *Figure 1*

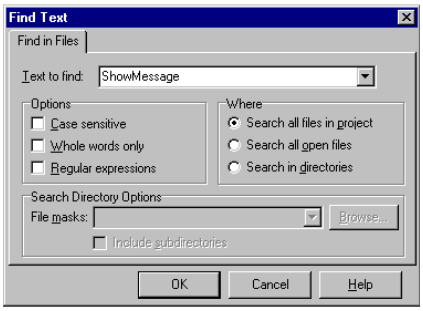
to make its constituent speed buttons merge in with the background (`TMediaPlayer`, however, does not).

The next thing you spot when you compile your first application is that the editor has a visible left-hand gutter (see Figure 1). No more will we all add breakpoints by mistakenly clicking on a very unobvious gutter in the first few millimetres of the editor. Whilst on the subject, a little tip for Delphi 1 or 2 users is to set the editor's right-hand margin to be at column 0. This makes the presence and position of the gutter much more evident.

This new gutter takes ideas from Turbo Debugger. It shows you which source lines have machine instructions generated for them, as well as showing where breakpoints have been set and displaying editor bookmarks.

A few other obvious IDE things to notice are some changed and some new menus. Delphi 1 had an `Options | Environment...` menu item. Delphi 2 changed this to `Tools | Options...` which was confusing since it wasn't clear what options it referred to. Delphi 3 addresses this by changing it to `Tools | Environment Options...`

This dialog's `Preferences` page has an option to configure the location of a shared Object Repository. The Repository defaults to the `ObjRepos` Delphi subdirectory but can be relocated elsewhere. This was always possible by adding a `BaseDir` string value into Delphi 2's



➤ *Figure 2*

`Repository` key in the registry, but it is now all formalised into the IDE.

Borland have bowed to Microsoft terminology and now all the experts in this product (and in C++ Builder) are called wizards (despite some Microsoft employees wincing at the term), so the `Database` menu has a `Form Wizard...` option.

The `Run | Parameters...` menu, which previously allowed you to specify command-line parameters for your application, now also allows you to specify a host application. This is to enable DLL debugging support, something that Delphi has been sorely lacking since its inception. Unlike Turbo Debugger, you debug an EXE or a DLL. You can't step from a Delphi EXE project straight into a Delphi DLL project, but it is much better than nothing.

The next noteworthy item on the agenda is on the `Search` menu: there is now a multi-file search option in `Find in Files...` (see Figure 2). It's really useful being able to search for something throughout all the

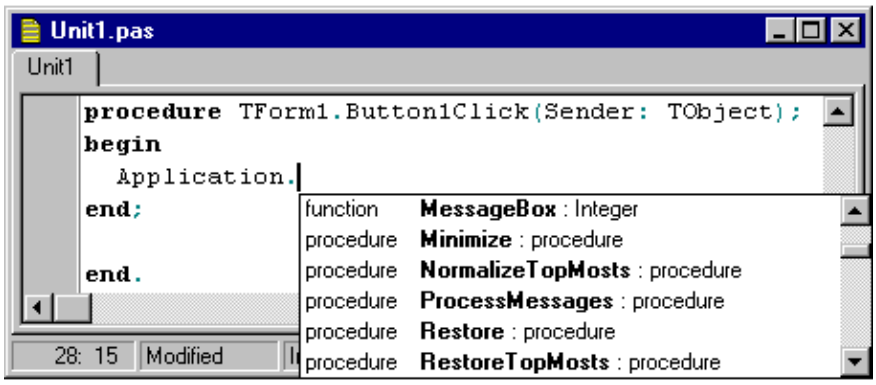
files in your project, or in a directory (with the option of including subdirectories). Rather pleasantly, the search operates in a background thread, so you can get on using the product whilst a large search takes place. The list of occurrences of the search text are placed in the editor message window indicating the file name and line number. Double clicking on these brings the file into the editor with the cursor on the appropriate line.

All the other new menu items are to do with new features such as Web application deployment and ActiveX control registration, type library viewing and COM interface modifications. We will get onto each of these subjects as we proceed through the tour.

The undocumented registry string value `EnableCPU` in the Delphi 3.0 `Debugging` key still works. If you set this to 1 a new `CPU Window` menu item appears next time you launch Delphi. This looks much better than it was in Delphi 2 (in fact it looks the same as the documented CPU window in C++ Builder) but the local menus generally don't do anything and I seem to get a lot of debugger crashes when it is active, which might explain why it is undocumented.

I'll just briefly mention that the `Win95` page of the component palette has now been renamed the `Win32` page, in honour of Windows NT 4 now offering support for these controls. There is also a new page called `Decision Cube` that we will come back to, but what shall we look at next? More editor features I think.

When you have a block of text highlighted, the editor allows you to drag it to a new destination, just as in Microsoft Word. Additionally, holding the `Control` key down before you start dragging allows text copying. It takes a while to get used to it and you may find you move text blocks accidentally when just intending to highlight a block of code. If you find yourself in the throes of moving a block you did not intend to, simply drag the cursor back to the block and you will drop it where it came from.



➤ *Figure 3*

Another drag and drop feature allows you to drag a file from Windows Explorer and drop it into the editor: the editor duly opens it. Okay, I admit that this feature has been around since Delphi 1 but I only found out about it the other day, so it feels new to me.

Code Insight

One of the nattiest additions in my humble opinion is referred to generically as Code Insight. This is in fact a set of editor time savers. Most of them operate by the environment constantly scanning editor source code and the contents of accessible units in a background thread. These features are similar to the sort of things that Visual Basic 5 offers in its environment.

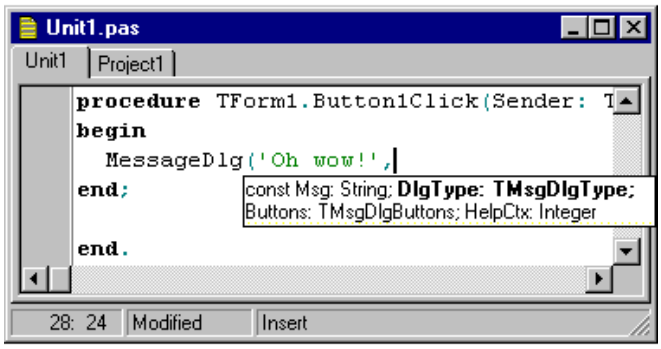
First off is Code Completion. If you type in an object name (even one of your own) and then type a period *[what us Brits like to call a full stop! Editor]* in preparation for typing a method or property name, a listbox magically pops up with all possible values available (see Figure 3). Select an entry and press `Enter` and it's typed in for you, with nice consistent case usage and never a typo to be found. If you want it to come up at any time on demand, press `Ctrl-Space`. The list can be sorted either alphabetically or in scope order.

Next on the block are Code Parameters. Type a procedure, function or method name (again, even one of your own) and the open parenthesis and a tooltip appears showing the parameter list as it appears in the routine's declaration: name, type and any modifier (Figure 4). The parameter that needs to be entered is emboldened and as you enter more parameters, the boldness moves along to the relevant parameters. It's way too cool! To get the tooltip upon demand, press `Ctrl-Shift-Space` (it won't show if it is not relevant).

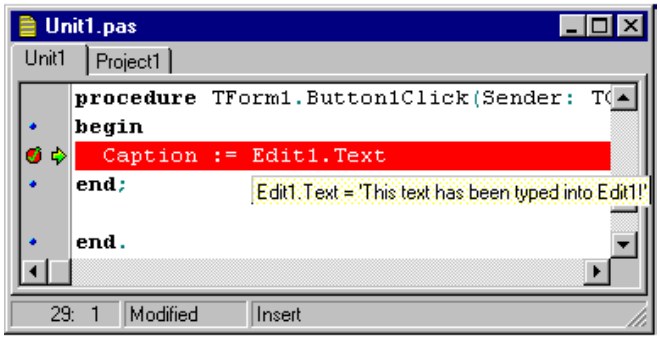
Incidentally, one word of caution is warranted here. Some Delphi 3 testers have encountered a problem when slightly mis-keying this keystroke combination. It's not a Delphi problem, but apparently Dell computers running Windows 95 lock up when you press `Ctrl-Alt-Space`, so be warned.

The third Code Insight feature is the tooltip expression evaluator. This concept was first introduced (as far as I know) in Microsoft Visual C++ and saves you mucking about with the Watch window (which, incidentally, by popular demand now has a stay on top option on its local menu). When the debugger is in control of the application you just put your mouse over some expression involving variables or properties and the value appears in a tooltip (see Figure 5).

Just as an aside, I heard a tooltip-related story involving an example of Danish humour. In the early days of the development of Delphi 1, while compatibility for the upcoming Windows 95 was being taken care of, Anders Hejlsberg (the chief architect of the product) saw a tooltip for the first time. The Dane saw a yellow area appear below the mouse pointer and announced to the closed R&D meeting that the mouse had just urinated... Okay, let's move along.



➤ *Figure 4*



➤ *Figure 5*

The last Code Insight feature is perhaps the one that has been most asked for one way or another: Code Templates. These can be set up on the Code Insight page of the environment options. When editing, you press `Ctrl-J` and get a list of code templates to choose from. The editor then types the template in. The pre-defined templates include a full class declaration with constructor and destructor, and an `if..then..else` statement, to name but two of the 22. If you have already started typing in a keyword, the list offered is reduced to those which will apply. These templates get stored in the DELPHI32.DCI text file in the BIN directory (the extension stands for Delphi Code Insight).

When you create your own templates you can insert a pipe sign (|) to indicate where the cursor should be left after the template has been typed in.

Project Options

In the project options dialog, as well as package options (they will be described later) there is a whole page for version information, as used in many commercial applications. This gets included into the project .RES file. On the `Directories/Conditionals` page you can specify a directory where compiled units are placed. Additionally, for those of you writing screen savers, OCXs, ActiveXs and other binaries that require a different file extension, you can set one up on the `Application` page. This adds a new compiler directive (`$Extension` or `$E`) into the project source file.

On the `Application` page, we can see that Delphi projects now get a revamped default project icon (see Figure 6)

One extra option on the `Compiler` page enables or disables assertions. This matches another new compiler directive, `$Assertions` or `$C`. Assertions are available in C and C++ and many people have tried to come up with a suitable implementation of them in Delphi. We now have an `Assert` procedure that can be used to test a Boolean expression: if it fails an `EAssertionFailed` exception gets generated, unless exceptions have been disabled, whereupon you will get a run-time error 227.

`SysUtils` replaces what would have been the normal dull assertion exception with a more interesting one that reports the source file and line number of the failure. You can also customise the assertion behaviour (for example to store the failure messages in an error log file) by assigning the address of a custom routine to the `AssertErrorProc` pointer. The procedure must be compatible with the following procedural type:

```
procedure (const Message,
  Filename: string;
  LineNumber: Integer;
  ErrorAddr: Pointer);
```

When you've used assertions to help thoroughly debug your application, you can remove all the assertion code from the EXE with the compiler option.

New Components

There are several new components I'll cover later, but some fairly independent new components added include the following.

➤ *Figure 6*

`TAnimate` encapsulates the Win32 mini-video `ANIMATE` control: the thing that does the annoying file copying animation in Windows 95 and NT 4. This supports all the predefined videos (via the `CommonAVI` property) and many external soundless AVIs (using the `FileName` property).

`TDateTimePicker`. This gives a convenient way of choosing dates and times. It requires the updated version of COMCTL32.DLL that ships with Internet Explorer 3.0.

`TOpenPictureDialog` and `TSavePictureDialog` are new extended versions of `TOpenDialog` and `TSaveDialog`. They have an area added on the side for previewing the selected picture. The `TOpenPictureDialog` is now used as the property editor for all bitmap type properties. We can kiss the crass *choose a picture and push the button before you see it* approach from Delphi 1 and 2 a fond farewell.

`TStaticText`, which is just like a label but has a window handle, supports being tabbed to and also offers various border styles.

`TCheckListBox` is a listbox where each item has a checkbox. The IDE uses these here and there as does the InstallShield set-up program.

`TSplitter` is a componentised version of the window pane splitter class from the Resource Explorer demo app from Delphi 2. It is basically the same as presented in Issue 18's *Delphi Clinic*, although the official one is simpler to use.

TChart. This is a component known generally as TeeChart licensed from TeeMach. The component editors allow you to order a full version from the original vendor, however on initial glances all that seems to be disabled are a few chart types. This component crops up three times on the component palette: once in its own right on the Additional page, once as a data-aware control (that doesn't even need a DataSource) on the Data Controls page and once as a new QuickReport component.

TDBRichEdit allows you to store formatted text into a database BLOb field.

TCoolBar. Implements the optionally transparent bands of controls found in Internet Explorer 3.0. Microsoft refer to this control as a "rebar". This also needs the updated COMCTL32.DLL.

TToolBar can be placed on a TCoolBar and caters for managing buttons and separator areas. This control also offers optional transparency. The point of this control is that a rebar's bands can only support one control. To get around this you place a toolbar in a band, where the toolbar can hold many buttons. You could also use some other container-type component such as a panel.

Component/RTL Modifications

All components that can edit text have Input Method Management (IMM) support to help write products with Asian language support. They have ImeMode and ImeName properties to specify Input Method Editors (IMEs). Also the TFont type has a new CharSet property, for the same reason. Interestingly enough Borland C++ Builder, which is based almost exclusively on the Delphi 2 VCL, also offers these new properties.

Additionally, SysUtils adds or modifies a whole raft of routines to support multi-byte character set (MBCS) operations.

In the same vein, there is a new string type, WideString. String is still defined to be an AnsiString, made of AnsiChars. A WideString is made up of WideChars, but in other respects operates with much the same behind the scenes automated memory management (except for reference counting).

The TGraphic base class of TBitmap, TIcon and TMetafile has had a few things added. Most notably there is a Palette property and an OnProgress event. The latter may be triggered during lengthy operations, such as loading a very large image from disk. Also, various graphic classes have thread-safe support and TBitmap now treats bitmaps as device independent bitmaps (DIBs).

Another nice graphics touch is that support for JPEG files has been surreptitiously added. Consider an application that has a TImage with code that calls

```
Image1.Picture.LoadFromFile
```

If you pass a .BMP filename or a .ICO, .WMF or .EMF filename then it will load fine. If you pass a .JPG file, it will not load. However now you can simply add the JPEG unit to your uses clause and it mysteriously does load the file up, if necessary taking care of half-toning and other technical graphics things. The Win32 import unit list has been increased. As well as the ActiveX unit, we also have ImageHlp (for pulling apart Win32 PE files), some units for the Microsoft and NetScape Internet Server APIs and MS Windows Internet extensions (ISAPI, ISAPI2, NSAPI and WinINet), a NetBIOS 3 unit (NB30), the Pen Windows unit is back (PenWin), common registry key string constants (RegStr), the Windows Shell objects for extending Windows Explorer (ShellObj) and Windows NT Services (WinSvc). Also, the RichEdit unit is about three times the size it was in Delphi 2.

Delphi 2 came with a demo in the IPCDemos directory that implemented some simple classes for representing a Win32 event, mutex and shared memory (via a memory mapped file). Delphi 3 now supplies a new RTL unit called SyncObjs that has classes for events and critical sections. It should be fairly easy to inherit from the base classes to support semaphores and mutexes as well.

All versions of Delphi have had a convenient routine available for IDAPI programmers. The Check procedure from the DB unit takes a return value from an IDAPI call and if it indicates anything other than success, an EDBEngineError exception is generated with an appropriate message by calling DbiError. Delphi 2 added a similar procedure OleCheck for OLE API calls which generates an EOLEError exception with an appropriate message.

Delphi 3 now adds a new one. SysUtils defines Win32Check which takes a Win32 API return. These are typically Boolean values. If the value is False, this routine calls RaiseLastWin32Error which generates an EWin32Error exception with a message obtained using SysErrorMessage and GetLastError. If the passed in value was True, Win32Check just returns the value.

Under some circumstances it will not be advisable to call Win32Check as some messages have a place-holder symbol which is intended to be substituted with something so this is not a foolproof solution (see Issue 16's *Delphi Clinic* item on *System error message* for an example of where this may crop up). However, with a bit of forethought and testing it should prove useful.

And last of all in this section is a real beauty. Historically, if you wanted to store a string in a string table resource, the approach involved having a unit dedicated to integer constants that could be used to refer to the strings. This was used by the .RC resource script file that defined the string table in terms of these constants. The RC needed to be compiled into a .RES file (possibly using BRCC.EXE) and needed an appropriate compiler directive ($R or $Resource) to get the resources into the EXE. The constant unit was also used anywhere in the project where the strings were actually required and the string was loaded from the resource with a call to LoadStr or FmtLoadStr.

All that is over now with the new resourcestring code section. It operates like a var, type or const declaration section and allows you to

define symbols with associated string values, rather like constants. However, all resource strings are stored as resources by the compiler (which also ensures that they all have different numbers). When a resource string is referred to, the compiler generates code to load the string from the resource from whatever module (package or EXE) that it happens to be in, using dedicated code in the `System` unit. Listing 1 shows the idea.

```
resourcestring
  SFirst = 'This is a resource string';
  SSecond = 'Easy to use, n''est pas?';
  SThird = 'Another one for good measure';
...
  ShowMessage(SFirst);
  Application.MainForm.Caption := SSecond;
  Application.Title := SThird;
```

➤ *Listing 1*

To find the resource ID of any resource string, typecast it into the `System`-defined `TResStringRec` record and check the `Identifier` field.

During compilation these get stored in a temporary binary resource file with the same name as the project, but with an STR extension and are then bound into the EXE along with any other resources during the link stage.

If you are a regular VCL source browser you will be aware that the `Exception` object's wealth of constructors were put in place principally for the benefit of the VCL authors. There are several which load resource strings: `CreateRes`, `CreateResFmt`, `CreateResHelp` and `CreatesResFmtHelp`. Since `resourcestrings` are now in use throughout the VCL you will find all previous occurrences of these constructors replaced with the non-resource versions: `Create`, `CreateFmt`, `CreateHelp` and `CreateFmtHelp`.

Component Templates

Delphi application developers often find themselves setting up groups of components in much the same way in many applications. Previously you had no choice but to set the components, properties and code up manually each time. Now Delphi helps automate this process. After painstakingly setting up your group of components, select them all in the Form Designer, choose `Component | Create Component Template...` and you can add a compound component straight onto the component palette. It won't be a true component, but who cares? It does the trick.

Whenever you want that group again, pluck it from the palette and the positioning, properties and all the event handlers are added instantaneously. If any of the event handlers previously referred to components in the group, then the new event handlers' code is generated bearing that in mind with references to the new components used instead.

All the details of these component templates are stored in the binary DELPHI32.DCT file in the BIN directory.

Packages

One of the main gripes that Borland Pascal and C/C++ programmers had about Delphi when it first came out was the size of the EXEs it generates. Each EXE had a large portion of the VCL compiled into it and so had an initial footprint of at least 180Kb. Database applications had a footprint of 330Kb. Delphi 2 reduced these a little. Admittedly people worry rather less these days, with the cost of disk storage dropping through the floor, but it is still less than desirable having the VCL duplicated through all your Delphi EXEs.

Delphi 3 helps avert this problem by introducing the concept of packages. These are DLLs with a DPL extension (for Delphi Package Library) that can have the various VCL units (and any others you care to use) compiled into them. You can compile your required units into as many or as few packages as you want, which can be marked as run-time only, design-time only or suitable for both. The Delphi component library is no more: design-time compatible packages take its place *[Hooray! No more corrupted libraries after failing to install a new component! Editor]*. Also, Delphi 3 itself is compiled using run-time packages and so despite the mass of new dialogs and environment features in the IDE, DELPHI32.EXE is only 500Kb larger than Delphi 2.

Delphi now comes with a whole host of packages that contain all the components from the component palette. The names of those installed into the environment are mostly prefixed with DCL and implement the various property editors and component editors required by the components, and contain the component registrations. The others are run-time only (you cannot install them into the environment) and implement all the components. The main one that gets referenced is VCL30.DPL and is just over 1.1Mb in size. This includes the basic run-time library code and basic VCL with no database or Internet components. Most of the database components are compiled into VCLDB30.DPL (585Kb).

The project options dialog has a `Packages` page that lists the design time packages that are installed. You can install and remove design-time compatible packages here, as well as see which packages installed which components. This is also where you decide if you wish to compile with package support or in the more traditional Delphi way.

If you do compile with packages, you can specify which packages should be considered for linkage (and by implication which units will be compiled directly into the EXE). The package requirements of the project get stored into a temporary Windows resource file with a .DRF extension as a custom resource type during compilation. They then get bound into the EXE during link time.

Despite packages being implemented as DLLs, you should not think of them as such. Instead, consider them as a linker option that

distributes code across binary modules but without changing the program semantics or organisation. You do not have to concern yourself with calling from the EXE to the DLL, you just call something in a unit and Delphi sorts out appropriate code to cross module boundaries where necessary.

So to the important information. If you compile a simple one form project with a button that calls `ShowMessage` it comes out at 195Kb without package support and 9.5Kb with packages enabled. This clearly means that an application suite that features many Delphi DLLs and EXEs can shrink in size markedly.

Packages are intended to help application maintenance and deployment. If you need to modify a piece of your application, you do not have to re-deploy the whole application, just the affected package. You also don't need to recompile the whole application.

However, not everyone is over the moon at the thought of packages pervading everyone's hard disks. They foresee that there can be potential mayhem when Borland bring out incremental releases of Delphi: the packages will all need new revisions. There may be a practical versioning issue to deal with, but hopefully Borland have considered this and know how to deal with the situation in an adult fashion. The consensus of many is that corporate-wide distribution of Delphi applications will be greatly helped by packages, but smaller outfits, like shareware authors, will probably not take advantage of them.

You can make a package by choosing the appropriate entry in the Object Repository. You specify a target file name and give a description (used principally by the IDE's packages dialog) and the file gets created in the IDE. A package source file has a .DPK extension and is displayed "visually" in a package editor (see Figure 7).

You can add units into the package and they are listed on the `Contains` page. A point to make clear now is that a unit cannot be placed into two packages that will

```
package PackageEg;
{$R *.RES}
...
{$IMAGEBASE $00400000}
{$DESCRIPTION 'A sample package'}
{$DESIGNONLY}
{$IMPLICITBUILD ON}
requires
  vcl30;
contains
  DCubeU;
end.
```

➤ *Listing 2*

be required (either explicitly or implicitly) by any one particular program. This is because all the packages brought in by a project live in the same name space.

Any required packages are added onto the `Requires` page. Various options like run-time only restrictions can be set and all these bits get stored in the package file as long format compiler directives. To see the underlying file of a package, right-click on the package editor and choose `View Package Source`. An example package file is shown in Listing 2 (with many of the compiler switch directives removed for brevity).

You can see that a package is indicated by the reserved word `package` and also has optional `requires` and `contains` clauses that get generated by the two pages on the package editor.

When the package is compiled it ultimately generates a .DPL file but also an intermediate .DCP file. This is a file that contains symbol information from each of the units that package contains. Additionally the package source file itself is compiled into a .DCU, and .RES, .DRF and .STR files are made as usual.

From the `File | Open...` menu item you can load up a package by opening the .DPK file (which you can edit) or the .DPL file (which you can't).

The new package support brings along various new compiler directives to help. `$DesignOnly` and `$RunOnly` are fairly self-explanatory and match the options available from the package editor, however they are only respected by the IDE. When a package is being installed the IDE will only accept it if it is marked for design-time use. Also, if

➤ *Figure 7*

it is not marked as being a run-time package it is automatically added to the run-time package list.

`$DenyPackageUnit` stops a unit from being packaged. `$ImportedData` disables the creation of imported data references for better memory access efficiency and effectively has the same effect as `$DenyPackageUnit`. `$ImplicitBuild` controls how often the package gets rebuilt.

Lastly `$WeakPackageUnit` is typically used to allow packages to contain units that require external DLLs but tries, where possible, not to include the unit in the .DPL file. Instead the .DCU compiled unit file is compiled directly into the EXE. This means the DLL does not necessarily have to be distributed with the .DPL. However if another unit in the package refers to the weak package unit, it will be compiled into the .DPL.

If this doesn't seem to make sense, remember the binary package file is supposed to have everything available in it that the constituent units declare: there is no smart linking. This means that all import declarations in any units contained in the package will be compiled into the binary package file and so there could be dependencies on many DLLs that aren't strictly required for all applications using the package. An example of a weak package unit is the PENWIN.PAS Pen Windows import unit. Since not all Windows systems have the Pen Windows DLL, it makes good sense to keep any explicit references to it out of any packages unless strictly necessary.

Package Collections

The Package Collection Editor (PCE.EXE) takes multiple compiled packages (.DPL files) and saves the list as a .PCE file (an .INI file). When you build the package collection you get a binary .DPC file (Delphi Package Collection, as opposed to .DCP, Delphi Compiled Package).

When installing packages with `Component | Install Packages...` (which, incidentally takes you to the equivalent of the project options `Packages` page) you can load a package (.DPL) or a package collection .DPC. If you choose a package collection, then you are presented with the Package Collection Installation Wizard which allows you to choose which of the constituent packages you want to install.

This facility allows component suite developers to supply many design-time package libraries in the convenient form of a single file.

Business Insight

This is the second use of the *Insight* term and refers to all the new database and decision support component architecture in the product. According to the slide in the slideshow I saw, it turns your data into actionable information. Well I don't know much about marketing, but I know what I like. And I do like the work done to the database support in the product. Without further ado, let's press on.

New VCL Architecture

There are several third-party libraries that allow you to write applications that talk to certain data formats without the use of the BDE. This gives a smaller distribution overhead, but was difficult to implement due to `TDataSet` having various BDE dependencies. It typically required modifications to the `DB` VCL unit.

In order to help third party developers, these dependencies have been removed from `TDataSet` and put in a new `TDataSet` descendant, `TBDEDataSet`, or in `TDBDataSet`, which now inherits from `TBDEDataSet`. This means that third parties can now simply derive from `TDataSet`. Additionally, all the BDE-related code has moved from `DB` to the `DBTables` unit to stop any application using `DB` from pulling in the BDE. Because of this overhaul, certain code that refers to `TDataSet` may need to be changed to refer to `TBDEDataSet` or `TDBDataSet`.

It would be nice to see an in-depth article on writing a BDE replacement using this new architecture that covers more than the Borland-supplied information. Would anybody care to take up the challenge?

There is default BLOb caching (which can be turned off using the `TDataSet` property `CacheBlobs`) to speed up scrolling through `TDBCtrlGrids` which have BLOb fields displayed. And yes, `TDBImages` and `TDBMemos` can now be placed on a control grid. In the version I am currently testing, `TDBRichEdits` cannot.

Database Explorer And BDE Administrator

The Database Explorer (or SQL Explorer as it calls itself in the Client/Server Suite) is now up to version 2.0. It sports many new features including the ability to drag stored procedures onto a form. It also allows you to edit various SQL objects, for example any view or stored procedure, InterBase generators and exceptions, Oracle packages and package bodies.

There is now a menu option for getting to the BDE Administrator (a much more user friendly application than the BDE Config app used to be). The BDE Administrator is based on the look and feel of the Database Explorer. In fact this is almost certainly a big code-sharing exercise since they both use the same INI-style configuration file, DBX.DBI. You can also invoke the ODBC Administrator from the Object menu.

One of the menus also allows you to set up transaction isolation level for the database connections.

New BDE

The new BDE is up to version 4 and now natively supports multi-byte character sets (MBCS support). Additionally it has two new *native* drivers for FoxPro and Access tables. These drivers make use of Microsoft's DAO connectivity, but unfortunately rely on you already having a properly licensed product that implements DAO, such as Microsoft Office 95 or 97.

If you feel that you need a native BDE driver that does not exist, and you also feel you can commit the time and patience to writing one, Borland may have something for you. The long-promised IDAPI driver SDK is now being made available by Borland. At the time of writing this is best obtained by speaking to Borland Developer Relations.

Distributed Datasets: N-Tier Computing?

Application partitioning is very fashionable nowadays. Delphi 2 made a token gesture in the

direction of splitting UI code away from data manipulation code with the data module. Delphi 3 allows full blown three-tier systems to be implemented using a new VCL feature called remote datasets.

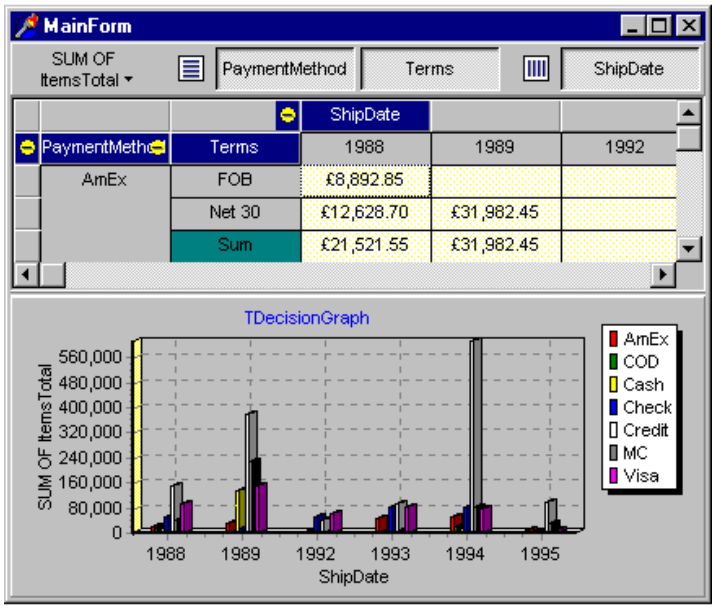
What this means is that an application can manipulate data in database tables using a dedicated dataset-based component but have no direct connection to the data. It also has no need for links to the BDE or any other database accessing technology. It talks to some data server application (or data broker as it is often referred to) using DCOM and the server application worries about the actual communication with the data in the database and applying appropriate business logic and data validation that may be necessary.

This was a reasonably easy mechanism for Borland to implement because of the improved COM support in Delphi 3 (which we will look at in depth next month) as well as the clean split in the database architecture between the dataset components and the BDE. There are three new components on the Data Access page of the component palette that are used to make a remote data server and a data client.

The TProvider component is used in the server application and packages data up from a BDE cursor via a TTable, TQuery or TStoredProc component into a data packet (the Data property) represented as a variant array of bytes for easy COM distribution. It contains both the data and appropriate metadata (schema information and constraint information). If you don't want to explicitly deal with data change errors using the provider's OnUpdateError event, or indeed use any other events it offers (such as BeforeUpdate and AfterUpdate or BeforeGetData and AfterGetData or OnDataRequest) then you don't need to use a provider object at all. The normal BDE dataset objects all have a Provider property that returns a TProvider of their own.

In the data client application a TRemoteServer is used to connect up to the server application. To

➤ *Figure 8*



connect to a server app on another machine, you have a ComputerName property. Also, there is a ServerName property which takes the server's ProgID (OLE class name). The TRemoteServer uses DCOM (distributed COM) to talk to the server app, but you can also use OLE Automation via the AppServer variant property.

In order for data aware controls in the client app to get access to the data in the server app, you use a TClientDataSet which connects, via the remote server component, to the provider in the server app. The TClientDataSet supports data editing and persistence. The server data packet is represented by a variant byte array property called Data (as it is in the provider). Any changes made to the data are stored in another variant array property called Delta.

To update the data in the server app the client app calls TClientDataSet.ApplyUpdates. This simply calls the ApplyUpdates method of the data broker's provider. Data errors can then be dealt with by the provider. The data server project (which is a COM server) needs to be registered in order for any clients to be able to talk to it.

Note that each connection to the data broker that refers to a remote data module will cause a separate instance of the data module to be created. If you wish to use a TDatabase component to set up the database connection then that must be placed on a normal data module. If you place it on the remote data module you will get problems due to multiple database objects using the same alias name.

If the client wants the server to return custom data it can call the server provider's DataRequest method like this:

```
ClientDataSet1.Data :=
  ClientDataSet1.Provider.DataRequest(
    'select * from orders')
```

The provider object's OnDataRequest event will trigger and your handler can process the request and therefore return the required data.

This approach of using COM to communicate the data means that the client application has no BDE overhead, all it needs is the DBCLIENT.DLL from the BDE directory: about 145kb in the field test I was using.

Incidentally, if you wish the data broker to be on one machine and the client to be on another machine running Windows 95, you will need to download DCOM for Windows 95 from Microsoft's Web site and install it.

Decision Cube

If you have a keen memory you may remember that one-time Borland product Quattro Pro for Windows had a nice data pivot cross-tab facility. Some nice developers in Borland were implementing similar

functionality for another project that seems to have fallen through, so it has been componentised for use in Delphi 3. However source code is not supplied in any version as Borland feel they may wish to use it in their own products in the future and don't want their algorithms to be known by potential competitive product writers.

Decision Cube refers to a set of decision support components that allow you to generate cross-tab graphs and charts to get views and summaries of your data from varying perspectives.

The `TDecisionCube` is a non-visual component that acts as a multi dimensional data store connected to a dataset. Typically the dataset is an SQL expression in a `TQuery` descendant structured in an appropriate way. The cube maintains an image of the data in such a way that it can perform various manipulations without having to re-query the original dataset.

In truth the cube's dataset is usually a `TDecisionQuery`, which has a component editor to help set it up quite easily, although you can use a normal `TQuery` or `TTable` if you like, with the required set-up done manually. The SQL expressions that can be used to populate a decision cube must have groups and summaries defined. A `TDecisionSource` component connects to a `TDecisionCube` and represents the current pivot state. The remaining three (visual) components connect to a `TDecisionSource`. The `TDecisionPivot` is used to identify what fields are used to display the summary view of the data: buttons allow you to open and close decision cube dimensions.

A `TDecisionGrid` displays the single or multi-dimensional data and a `TDecisionGraph` (which is based on a `TChart`) shows the data graphically. The graph and grid component automatically redraw themselves when the dimensions are changed.

The components have been designed to allow drag and drop operations to change which order the fields or decision cube dimensions are in, and to allow you to drill down from a view where summary data is displayed only for a particular value into general summary data and then full data view. A simple SQL expression such as:

```
SELECT ShipDate,
  PaymentMethod, Terms,
  SUM(ItemsTotal)
FROM ORDERS
GROUP BY ShipDate,
  PaymentMethod, Terms
```

can give such output as displayed in Figure 8 with practically no extra set-up required.

And Finally

That takes up all the space I am allowed this month. Next month we will look at all the new COM, OLE, ActiveX and Web-based features that are new in Delphi 3. Until then...

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