Beating The System: IDE Secrets, Part 1

by Dave Jewell

Here's a little experiment you might like to try out. Create yourself a new application in the Delphi IDE (I'm assuming that you're using Delphi 5, by the way: your mileage may vary with earlier incarnations), and then go to the `Packages` tab on the `Project | Options` dialog. Click the `Build with Runtime Packages` checkbox to ensure that you end up with a 'packaged' version of your application. Next, add the package name `DSNIDE50` to the list of referenced runtime packages immediately below the aforementioned checkbox. You will find that your application compiles and runs as normal.

If you now add the unit name `DockForm` to your `uses` clause, the program will compile and run as before. Next, try changing the ancestor class of your form window from the usual `TForm` to `TDockableForm`. Again, the compiler will build your application without complaint, *but* this time you will get an access violation when you execute it.

So what's going on here and what on earth is all this stuff?

IDE-Only Packages

Despite frequent newsgroup rumours to the contrary, the Delphi and C++Builder IDEs are both written in Delphi. Up until Delphi 3 (if memory serves), the development environment was a single chunk of monolithic code. Having said that, it did still require the presence of a number of extra DLLs, notably the compiler (which has always been written in C), together with the debugger, and a couple of smaller DLLs which implement the core functionality of the text editor engine and keyboard processing. To put this another way: prior to Delphi 4 the IDE was an unpackaged executable, but from Delphi 4 onwards the IDE started using packages in a big way.

Some of the packages used by the IDE are the ones we all know and love, such as `VCL50` (again, I'm sticking with Delphi 5 for the sake of argument) but others are more private packages that are designed specifically for the benefit of the IDE itself. Chief amongst these is `CORIDE50`, which implements much of what we perceive as being the IDE proper. This particular monster weighs in at nearly 4Mb. In the past, Borland R&D have told me that splitting up the IDE into separate packages makes it easier for them to work on the development system in a team environment, but (from my perspective) it also has the not inconsiderable benefit of making it easier to poke around and see what's going on 'under the hood'!

A number of Delphi's internal packages are somewhat less secret than others. If you have done much work with packages, you will know that, when building a new package, the development system spits out not only the .BPL executable, but also the corresponding .DCP file. A .DCP file is essentially a concatenation of all the individual .DCU files in the package, and having the .DCP file to hand makes it possible to write code which accesses the goodies inside a package, *without having access to the package source or the individual DCU files which make up the package*. This is a very important point!

Now, consider the aforementioned `CORIDE50` package. Because Delphi 5 doesn't ship with the corresponding CORIDE50.DCP file, it is not possible to simply add `CORIDE50` to a new project in the way that I've just demonstrated with `DSNIDE50`. Needless to say, the DSNIDE50.DCP file is included with Delphi 5. You can find it in your `DELPHI5\LIB` directory, where you will also find numerous other .DCP files.

As you will have guessed by now, `DockForm` is the name of one of the units inside `DSNIDE50` and `TDockableForm` is one of the various classes exported from this unit. The reason our sample project crashed is not surprising: all the code in these IDE-only units and packages was written specifically to be used from within the Delphi IDE, if you try and execute it within the context of a standalone application then things will certainly screw up.

A Treasure Chest For Expert Writers

The fact that `DSNIDE50` and its brethren cannot be used from an application matters not a jot if you are in the business of creating Delphi IDE experts and add-ins. If this is the case, then I think the remainder of this article will prove quite an eye-opener for you. In reality, these undocumented packages contain a wealth of goodies for those who want to augment the functionality of the IDE.

As an example of what I'm getting at, try pointing your web browser at www.puthoon.com/ideDock/ideDock.html. This is the home of Mr 'Puthoon' (nee Python) who has put together a set of Delphi units which make it possible to create add-in Delphi experts with forms that dock with the 'regular' IDE windows (the Object Inspector, Project Manager and so on) in the usual way. Much of this code derives from GExperts and the work of others, such as Stefan Hoffmeister.

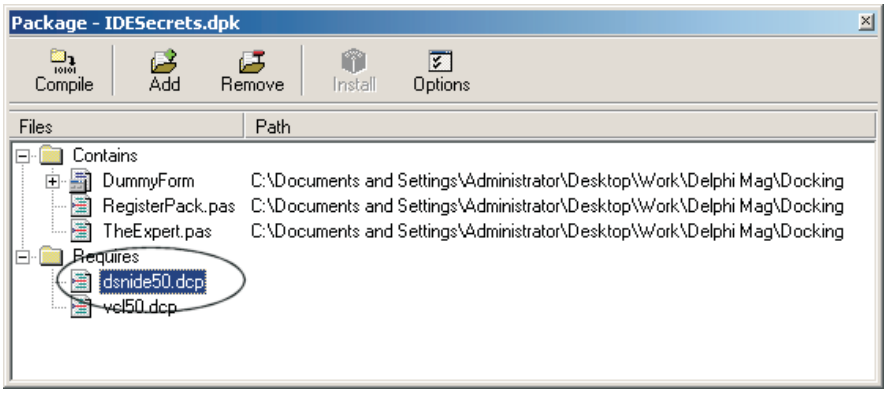
However, if you peruse this code, you will quickly discover that the programming techniques used are rather... umm... unconventional, to say the least. To cut a very long story short, all this stuff

➤ *Listing 1*

```
procedure InvokeDummyExpert;
begin
  TDummyExpert.Create(
    Application).Show;
end;
```

works by doing really ugly things such as patching VMT tables to fool the IDE into thinking that your new expert form is derived from the same ancestor that's used to implement IDE dockable windows. It works, but it represents a rather dirty hack of truly gargantuan proportions. Surely there must be a better way of doing the job?

There certainly is. Hopefully, by now you are beginning to see where I'm going. Surely it should be possible to use DSNIDE50.DCP (and others) to write IDE plug-ins which integrate into the IDE just as naturally as if they'd been written by Borland themselves?



➤ *Figure 1: Having the relevant .DCP file is the key to easily accessing the undocumented units contained within the DSNIDE50 package. Just add it to your package project and away you go.*

Painless Dockable IDE Windows

To try this out, I created a new package and wrote a minimalist Delphi expert derived from `TIExpert` in the usual way. This is a 'standard' expert, which means that it simply appears as a menu entry, `IDE Secrets Demo Expert`, on the development system's `Help` menu. The business end of the expert is simply this:

```
procedure TDemoExpert.Execute;
begin
  InvokeDummyExpert;
end;
```

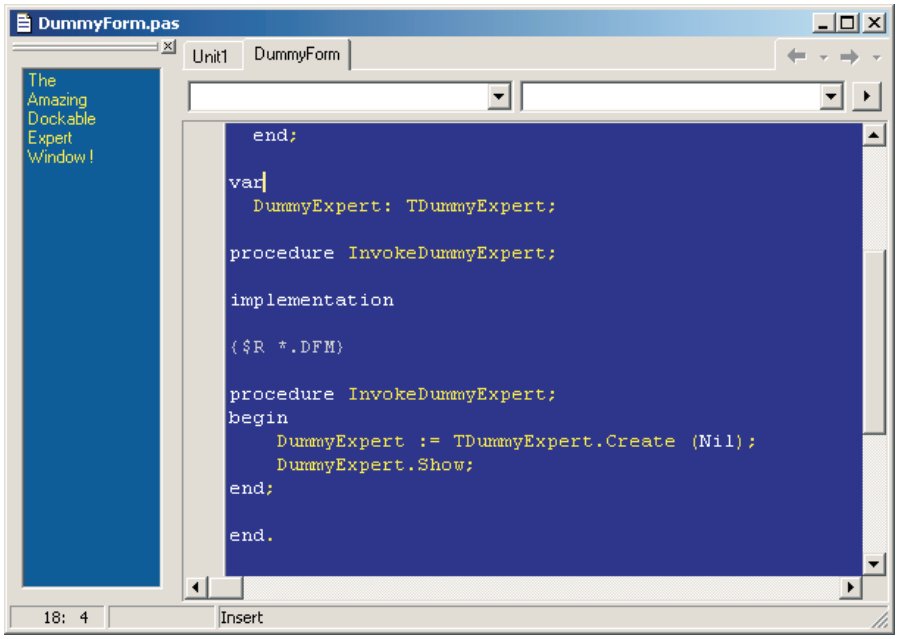
The `InvokeDummyExpert` call is defined in another unit. It merely creates an instance of a form class, `TDummyExpert`, and ensures that it's visible: see Listing 1.

All standard stuff so far. Now here is the cunning part: with the package editor window open in the IDE (see Figure 1) I right clicked on the `Requires` node of the file list, selected `Add` from the popup menu, navigated to the location of my DSNIDE50.DCP file and added it to my new expert package. Finally, inside the form unit, I added `DockForm` to the `uses` clause and changed the ancestor class of `TDummyExpert` to `TDockableForm`. After recompiling the expert and invoking it from the `Help` menu, I was rewarded with Figure 2, a nice dockable form that looks and feels just like a 'native'. I think you will agree that doing things this way is a *lot* less hassle!

To be fair to Messrs Puthoon, Hoffmeister, etc, it's got to be admitted that this technique will really only work for Delphi 5 and later. Prior to Delphi 5, Borland didn't ship DSNIDEXX.DCP as part of Delphi and therefore you were reduced to chasing VMT table addresses.

Of course, anyone who has spent some time working with Eagle Software's deservedly popular CodeRush 5.0 system will know that this also allows you to create new CodeRush plug-ins that dock with native IDE windows in the same way. I suspect that Mark Miller (creator of CodeRush) does this by creating `TDockableForm` descendants, just as I've done here, but I haven't investigated this in detail.

➤ *Figure 2: Now wasn't that easy? A fully paid-up, dockable expert window that looks and feels just like the stuff that's built into the IDE itself. A lot less hassle than VMT hacking...*



How It Works: The IDE Desktop

So, if we can get IDE dockable forms just by changing the ancestor to `TDockableForm` and linking against `DSNIDE50`, then what other wonderful goodies are lurking inside this package? The answer is a great deal. In this section I'm going to take a more detailed look at `TDockableForm`.

As we have already seen, `TDockableForm` is defined inside the `DockForm` unit. This class, in turn, is derived from `TDesktopForm` which

is defined inside another internal unit called DeskForm. TDesktopForm, you will be glad to know, is a direct ancestor of TForm. Thus, the overall arrangement looks like this:

```
TForm (FORMS.PAS)
  TDesktopForm (DESKFORM.PAS)
    TDockableForm (DOCKFORM.PAS)
```

Since we're all familiar with TForm, let's begin with TDesktopForm and the DeskForm unit, seeing what extra functionality they bring to the party.

As the name suggests, the DeskForm unit is all about the desktop: not the Windows desktop, but the IDE desktop! The primary job of this unit is to add the necessary functionality to a TForm descendant such that it works transparently with the Delphi desktop. If you've used this feature much, you'll know that the Delphi IDE has the ability to save an arbitrary docking arrangement of IDE windows, including the size and position of each form. Wouldn't it be great if you could write add-in experts which work in exactly the same way?

The good news is that it's very easy to do this, because all the required code is already built into DeskForm. But before we can do that, we need to have a greater understanding of the IDE's desktop architecture. To begin with, you need to register your form as a fully paid up member of the desktop community. This is typically done in the initialization clause of your form unit, as I've shown in Listing 2.

In this particular example, I've shown the actual code snippet that the built-in property inspector uses when registering itself with the desktop system. The RegisterDesktopFormClass is another internal routine, defined inside another unit called DeskUtil. This unit also lives inside the DSNIDE50, so you can just add it to your uses clause as normal.

This is another reason why this month's musings are specific to Delphi 5.0. In Delphi 4.0, the RegisterDesktopFormClass was implemented inside a unit called Desktop rather than DeskUtil. A lot of rearrangement of the internal routines took place between 4.0 and 5.0, and it's my hope that Borland's decision to ship DSNIDE50.DCP is symptomatic of an eventual plan to officially document all of this stuff. Maybe in Delphi 6.0? Here's hoping.

If you want to see what the function prototype for this routine looks like, here it is:

```
procedure RegisterDesktopFormClass
  (AFormClass: TDesktopFormClass;
  const Section: String;
  const InstanceName: String);
```

The TDesktopFormClass type is defined inside DeskForm.pas like this:

```
TDesktopFormClass =
  class of TDesktopForm;
```

In other words, you can pass any derivative of TDesktopForm as the first parameter to the routine. The Section and InstanceName parameters work somewhat like the Section and Name parameters used by .INI files. To see why this should be so, try saving a desktop arrangement using the IDE and you should end up with a file that has the extension .DST. If you open this file using a standard text editor, you will see that it is indeed structured just like an .INI file. Figure 3 shows a portion of a desktop file: just the part that's specific to the property inspector window.

Unless you plan to support multiple instances of your IDE expert, you should set the InstanceName and Section parameters to the same string. Be sure not to use a section name that already exists. If you do, things will go pear-shaped very quickly! Most of the existing section names have obvious values such as PropertyInspector, AlignmentPalette, CodeExplorer and so on. You can discover more of them by inspecting the contents of a saved desktop file.

Something else you need to be aware of is the way in which TDesktopForm introduces FormCreate and FormDestroy handlers within the DeskForm unit. You don't need to worry too much about what's going on in these handlers, but if you implement FormCreate and/or FormDestroy handlers of your own, then you should ensure that the inherited handlers get called in the usual way, as shown in Listing 3.

Oh, yes, I didn't mention DeskSection, did I? This is a readable and writeable string property of TDeskForm. Again, it needs to be set to the section name of your form class. This should be done *before* you call the inherited FormCreate routine. After calling FormCreate, you've got the opportunity to perform any other expert-specific initialisation. For example, you might set the form's HelpContext property here and (if things have been properly integrated with the Delphi help system) your own expert-specific help info will appear if the developer clicks F1 while your expert has the focus. As an example, try setting the HelpContext property to $578 and you'll see the help information for the property inspector.

```
initialization
  RegisterDesktopFormClass(
    TPropertyInspector, 'PropertyInspector', 'PropertyInspector');
end.
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TDummyExpert.FormCreate (Sender: TObject);
begin
  DeskSection := 'DummyExpert';
  Inherited FormCreate (Sender);
  ... other initialisation here ...
end;
```

Further Desktop Integration

TDesktopFormClass defines a number of virtual routines, one of which is SaveWindowState, defined like this:

```
procedure SaveWindowState(
  Desktop: TMemIniFile;
  isProject: Boolean);
  virtual;
```

At the point at which this routine gets called, the IDE is trying to save the current desktop configuration. The `Desktop` parameter is obviously a reference to the target file whereas `isProject` indicates whether or not (I think!) a project is active at the time the desktop is saved: this may or may not be relevant to your expert.

It's very useful being able to override `SaveWindowState` in order to save your own form-specific information as part of the desktop file. For example, Listing 4 shows a sample of what the property inspector window does.

As you can see, it simply calls the inherited `SaveWindowState` routine and then uses the `TMemIniFile` variable in the normal way to write an integer variable to the file. This is saved with a key name of `SplitPos`. As you might guess, this corresponds to the horizontal position of the vertical dividing line that forms an integral part of the property inspector. I have (rather tastelessly I'm afraid) highlighted the corresponding entry in the desktop file in Figure 3.

`PropertyList` is a property of the `TPropertyInspector` class itself and equates to the specialised listbox control that's used to display properties and their corresponding values, whereas `Middle` is a property of that control, and obviously gives us the current divider position.

The other side of the coin is another virtual routine, `LoadWindowState`, which is defined like this:

```
procedure LoadWindowState(
  Desktop: TMemIniFile);
  virtual;
```

Using these two routines, `LoadWindowState` and `SaveWindowState`, you can easily save your own custom window state info as part of the desktop file.

TDockableForm

There's more that could be said about `TDeskForm`, but let's give it a break for now. Instead, we'll take a look at `TDockableForm` which, you will remember, is a descendant of `TDeskForm` and adds docking capabilities to the class.

If you examine all the dockable menus inside the Delphi IDE, you will see that they all have a right click context menu which incorporates a `Dockable` item. This is checked or unchecked according to whether or not the form is in a dockable or undockable state. Selecting the menu item toggles from one state to the other. In order to do the job properly, we would obviously like to emulate this behaviour. How is it done?

If you could peek at the declaration for `TDeskForm`, you would see something like Listing 5.

The key to toggling the 'dockability' of your expert form is to add a new popup menu to the expert form, create a new menu item (say, `Dockable1`) and link its `Action` property to the `DockableCmd` field like this:

```
Dockable1.Action :=
  DockableCmd;
```

You can conveniently place this statement into your form's `FormCreate` routine as previously discussed. With this done, the menu item will automatically be named `Dockable`, it will be checked or unchecked as appropriate, and the right thing will happen when the item is selected. If an expert form is currently docked to another window, it will snap away as soon as the `Dockable` menu item is unchecked, just like you'd expect. In a similar way, you can implement `Stay on Top` and `Zoom Windows` functionality in your form expert's context menu merely by creating more menu items and assigning their `Action` properties to the other two fields shown above. Easy peasy!

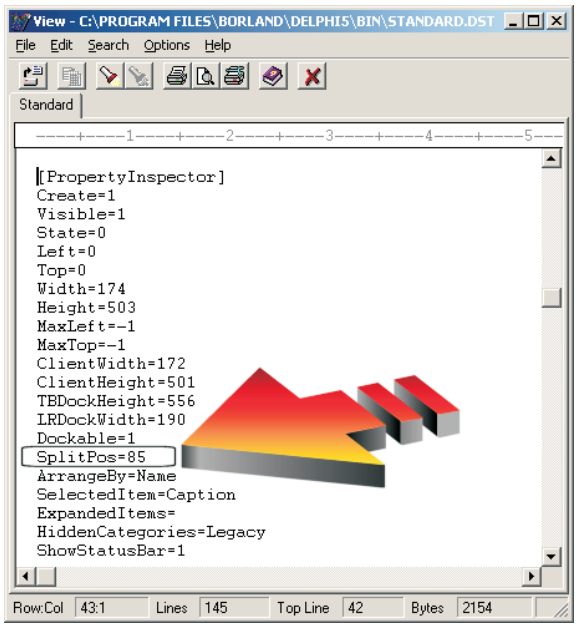
If, for whatever reason, you don't like working with action lists,

```
procedure TPropertyInspector.SaveWindowState(
  Desktop: TMemIniFile; isProject: Boolean);
begin
  Inherited SaveWindowState (Desktop, isProject);
  Desktop.WriteInteger (DeskSection, 'SplitPos', PropertyList.Middle);
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
type
  TDockableForm = class(TDesktopForm)
  published
    DockActionList: TActionList;
    DockableCmd: TAction;
    StayOnTopCmd: TAction;
    ZoomWindowCmd: TAction;
    ... more stuff ...
```



➤ *Figure 3: Here's part of a desktop configuration file as saved by the IDE. As you can see, it's essentially a specialised .INI file. The highlighted line shows how it is possible to save custom window-specific information out to the desktop file (see the text for more info).*

or you would rather toggle a form's dockable state in some other way, you can directly assign to the `Dockable` property of the form. It's a read/write `Boolean` property.

This would probably be a good point at which to mention menus. Earlier, I stressed the importance of remembering to call the inherited `FormCreate` and `FormDestroy` handlers whenever you override these event handlers. There are several reasons for this, but one of them relates to the menu handling system within the IDE. As you might imagine, you have potentially got numerous popup menus associated with different windows, each of which has its own set of keyboard shortcuts. In order to tie everything together, the `FormCreate` handler of `TDesktopForm` checks to see if the form has got an associated popup menu. If so, the popup menu is automatically registered with a centralised menu manager.

Implementing ToolBar Forms

Of course, there are other descendants of `TDockableForm` which are equally interesting.

If you add the `DockToolForm` unit to the `uses` clause of your form unit, then you can create a new form-based expert which derives from `TDockableToolbarForm`. As the name suggests, this is a dockable form which also incorporates a toolbar. The familiar project manager window is an example of such a form, the package editor is another.

If you have used the project manager or package editor very much, you will know that these IDE windows have a nifty (but not instantly obvious!) feature: you can hold down the mouse over the bottom edge of the toolbar area and then drag the mouse up or down to create a large or small icon toolbar.

Strictly speaking, this is a bad choice of terminology, since the size of the toolbar icons does not actually change, as shown in Figure 4. What happens is that in 'small icon' mode the button captions are not drawn beneath each glyph. You'll be pleased to know that, as

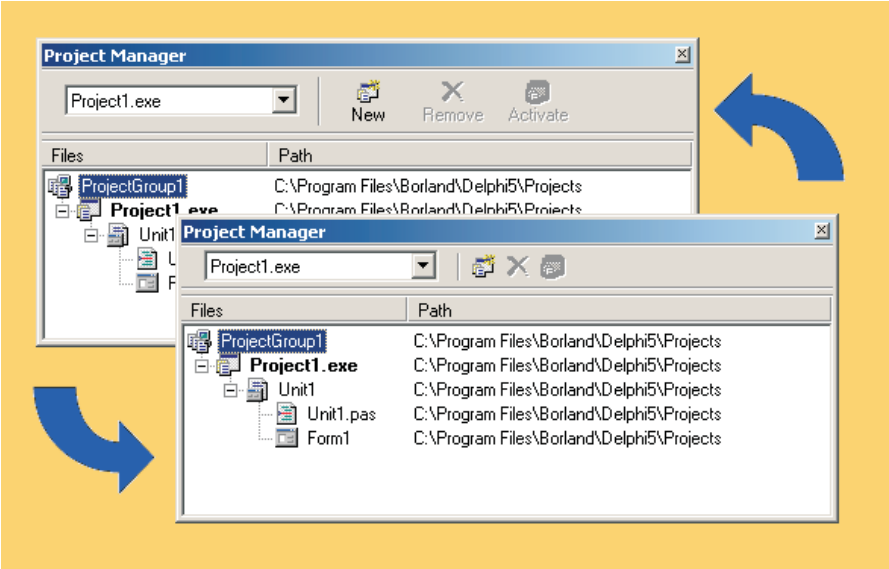
```
procedure TDummyExpert.FormCreate(Sender: TObject);
begin
  DeskSection := 'DummyExpert';
  Inherited FormCreate (Sender);
  ToolBar1.Images := ToolImages;
  // Create some buttons in the toolbar as per Package Editor
  with TCommandButton.Create (ToolBar1) do begin
    Parent := ToolBar1;
    Left := 4;  Top := 0;  Width := 56;  Height := 36;
    Action := CompileAction;
    Images := ToolImages;
    ShowCaption := True;
  end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
uses
  Classes, IDECommandButton, IDEMenuAction, IDEDockCtrls, IDEDockPanel;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('IDE Internals', [TCommandButton, TPopupAction,
    TTabDockPageControl, TDockPanel, TEditorDockPanel]);
end;
```

➤ *Figure 4: The Project Manager and Package Expert are examples of built-in form classes that implement a 'squeezable' toolbar that can be dragged into either a large or small configuration.*

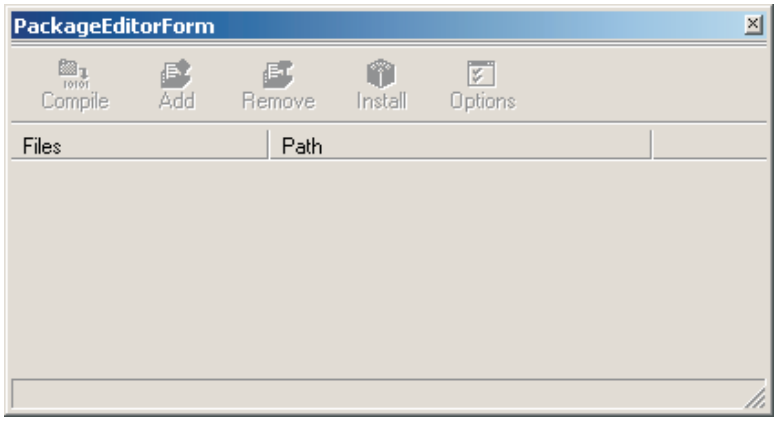


before, this functionality comes as part of the `DockToolForm` unit.

To demonstrate how this all works, I created another dummy expert form derived from the `TDockableToolbarForm` class. I also used a little sleight of hand to extract the two `TImageList` components that belong to the package editor from the Delphi IDE and incorporate them into my new form. Having done this, it was possible to add new buttons to the (inherited) toolbar control easily with the code in Listing 6.

A compiled version of this dummy expert, which is called IDESECRETS.BPL, is included on this month's companion disk, along with complete source code. You install the .BPL file into the Delphi IDE in the usual way, whereupon you will see a new entry on the Delphi `Help` menu. Click this item and the dummy expert will appear. Figure 5 shows what you should expect to see. You can resize the toolbar just like the project manager or package editor, and of course it's a dockable form too. Yes, the actual buttons are disabled but, hey, this is a dummy expert! It's up to you to add your own code to do something useful here.

If you consult the source code, you will notice that I have also duplicated the action list



➤ *Figure 5: Here's my clone of the Package Editor form. Although the toolbar buttons are disabled in this example, the toolbar works as advertised, and it is fully dockable. It would be a lot of work to get this working if we weren't making use of internal IDE routines.*

component from the IDE itself. This action list implements various actions that relate to the 'real' package editor. It's included purely for illustrative purposes.

If you're sharp eyed, you will also spot the fact that I'm using a hitherto unmentioned component, `TCommandButton`, to create the buttons that appear in the dockable toolbar form. This button is defined in the `IDECommandButton` unit which also happens to be part of `DSNIDE50`. This button control is derived from `TSpeedButton` and adds some extra functionality, most notably in terms of enhanced resizing and layout control, which of course is exactly what's needed for us in the dockable toolbar form scenarios that we've just been looking at.

More Dastardly Deeds

When I discovered that there were a number of interesting internal VCL controls inside `DSNIDE50`, I realised that these controls could be made to appear on the IDE's component palette simply by writing a do-nothing package whose only job is to register the components in the usual way, via the `RegisterComponents` routine. This is illustrated in Listing 7.

Sure enough, this works as advertised, and you end up with some interesting new components on your palette. It goes without saying that none of this stuff is of any use for writing ordinary applications; it's specific to the creation of experts and add-ins. Incidentally, if this sort of thing turns you on, you might like to know that there are lots more VCL controls buried inside the VCLIDE50.BPL package. The bad news is that Borland don't provide a .DCP file for this package, but I'm working on it ☺.

In next month's column I'll continue this investigation of Delphi IDE internals and we'll peer even more deeply into what's available to the aspiring expert author.

Dave 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 TechEditor@itecuk.com

➤ *Figure 6: Looks like an ordinary collection of Delphi components? These are internal IDE controls that have been exposed to our gaze through the simple expedient of writing a design-time package that registers them on the Component Palette.*