Distributed Application Development With Packages

by Brian Long

These days, a common reaction to hearing the term *distributed application* is to think of an application built using DCOM or CORBA, spread across a number of machines. This is understandable, as a DCOM or CORBA application is indeed distributed over potentially many different machines, with the various sections communicating using some appropriate protocol.

But one of the joys of any spoken language is the ability to interpret things differently depending upon context. This article looks at a different interpretation of distributed application development: developing an application designed to run on a single machine but distributed over several binary files. Clearly this is not a new subject at all, as applications have been split into executables and DLLs for years, but we will look at an alternative spin on the situation.

In this case, we will be building an application using Delphi *packages*, and the point is to show how it is possible to build the application in such a way that it can be extended in functionality *after having been deployed* by the developer(s). We will define guidelines for writing add-in modules. We will also see how the original developer can just open the doors to allow third party developers to modify the program in ways previously unimaginable with ordinary DLLs.

Since the discussion revolves around packages, and these were introduced to us by Delphi 3, the scope of this article is limited to Delphi 3 or later. Rather than delving into the subject and principles of using packages in your development cycle, I will instead refer you to *Under Construction: Delphi 3 packages* by Bob Swart and Chad Z Hower in Issue 23, which gives an excellent grounding in the subject. Additionally, I gave an overview of Delphi's package technology in *The Delphi 3 Novelty Store: 1* in Issue 20.

Just before moving on, allow me to make a couple of points. Some time after Delphi 3 was released Borland uploaded a sample application to their website showing a simple use of dynamic package loading. At the time of writing this can be found at

```
www.borland.com/devsupport/
  delphi/download_files/
  pkgdemo.zip.
```

The package mechanism provides a superb opportunity for dynamically extending the functionality of an application, and I have seen nothing in print on this aspect of package usage. Or at least I hadn't until I had finished this article and found Steve Scott's coverage of writing plug-ins in the March/April 1999 issue of the UK-BUG newsletter. Because of this lack of coverage I suspect this powerful facility is very much under-used.

Before looking in detail at the subject, a good part of this article will attempt to establish some groundwork on the subject, looking at an existing application for familiarisation. Only then will we focus on a sample implementation.

One final thing before proceeding. Whilst these techniques will work with Delphi 3 and Delphi 4, they also work perfectly well with C++Builder 3 and 4. In fact, packages written in either C++Builder 4 or Delphi 4 can be used by applications written in either product.

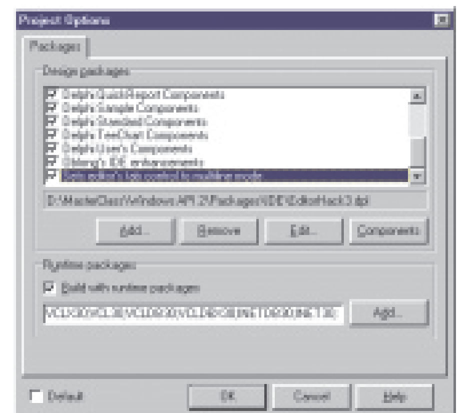
A Fine Example

To get an idea of some ways that we could proceed, let's consider Delphi itself. As you probably know, Delphi is a splendid example of an application written in Delphi. Also, as you should know, Delphi is very extensible and customisable. In both Delphi 3 and 4, which (as I write) are the only versions that support development of packages, you extend the environment's capabilities by writing code that is installed in *design-time* packages. Design-time packages are installed into the IDE using the dialog invoked through `Component | Install Packages...` (see Figure 1).

These design-time packages are distinguished from *runtime* packages by the fact that Delphi explicitly loads design-time packages, whereas runtime packages are typically loaded implicitly, either by your own applications or by design-time packages. So, packages used *directly* by Delphi are design-time packages. Packages used by your app, or *indirectly* by Delphi, are runtime packages.

For another look at the distinction between design-time and runtime packages, you can check the *Delphi Clinic* from Issue 30. However, in brief, runtime packages typically implement functionality required by your application to operate. This functionality could be functions, procedures, classes, forms or whatever may be compiled into a unit. Design-time packages tend to implement

➤ *Figure 1: Delphi's IDE dynamically loads packages.*



functionality (in the Delphi IDE) that is more desired than required. You can ask Delphi to load up a design-time package whenever you like, you can also ask it to unload it at any time.

Delphi Design-Time Packages: What Do They Do?

So the point being made above is that design-time packages can be used to add functionality that is not so much required by the program to operate, but is potentially useful to have. Like what?

Okay, there is a list of things that packages can do to Delphi, typically involving installing something. All the documented and advertised facilities are based upon a registration routine. As the *IDE Dissatisfaction* entry in the *Delphi Clinic* from Issue 43 discussed, any unit that goes into a package and needs some initialisation or registration code to be executed can implement a routine called `Register`. As long as this is declared in the `interface` part of the unit, Delphi will locate the routine when the package containing the unit is loaded, and call it.

And now onto the list of the more common things we can do in a design-time package. This will allow us to see some of the techniques used by Delphi to manage the things that get installed. These techniques form the basis for some of the mechanisms for our own distributed and extensible application development framework.

Installing Components

Firstly, the main thing that most people know how to do is to write and install a component class. The class type is passed to the `RegisterComponents` routine (or more rarely, the `RegisterNoIcon` routine) in the `Classes` unit in order for Delphi to be notified of its existence. The `RegisterComponents` (or `RegisterNoIcon`) call is made from within the `Register` routine.

These two component registration routines are both implemented in the `Classes` unit, but do very little (Listing 1). Assuming an appropriate non-local procedural variable is non-`nil`, the parameters are passed straight through, otherwise an exception is raised.

```
resourcestring
  SRegisterError = 'Invalid component registration';
var
  RegisterComponentsProc: procedure(const Page: string;
    ComponentClasses: array of TComponentClass) = nil;
  RegisterNoIconProc:
    procedure(ComponentClasses: array of TComponentClass) = nil;
procedure RegisterComponents(const Page: string;
  ComponentClasses: array of TComponentClass);
begin
  if Assigned(RegisterComponentsProc) then
    RegisterComponentsProc(Page, ComponentClasses)
  else
    raise EComponentError.Create(SRegisterError);
end;
procedure RegisterNoIcon(ComponentClasses: array of TComponentClass);
begin
  if Assigned(RegisterNoIconProc) then
    RegisterNoIconProc(ComponentClasses)
  else
    raise EComponentError.Create(SRegisterError);
end;
```

➤ *Listing 1*

If you were to call either of these two routines in a normal application, the procedural variables would have their default `nil` values, and you would get an exception. When a design-time package calls the routines within the IDE, no exception occurs. This is because some internal IDE unit has set the procedural variables to refer to some internal IDE routines. Presumably the routine assigned to `RegisterNoIconProc` adds the various classes to an internal list of maintained component types (instances of which can be accessed from the Object Inspector). The `RegisterComponentsProc` will do the same, but additionally do the extra work required to get the components showing on the component palette.

Exactly the same approach is used when installing other things into the IDE, such as custom modules or property mappers. The details of these things themselves are not important, as we are only looking at how the IDE manages to keep track of installed items.

Installing Property Editors

When you write a new component class and want to be nice to the potential users of your components, you write property editor classes as well, inheriting from one of the `TPropertyEditor` class descendants defined in the `DsgnIntf` Open Tools API unit. This class, as well as details of the class whose property you are supplying an editor for, plus information on that property are passed to `RegisterPropertyEditor`, whose implementation is in Listing 2.

You can see that in this particular case, there are no internal subroutines at work. Instead, the unit defines a list and the `RegisterPropertyEditor` routine fills it with allocated records. This unit, `DsgnIntf`, is contained within the VCL30.DPL or VCL40.BPL package, which itself is a required package of both the Delphi IDE and any package that you write yourself. In other words, your registration code causes records to be added to a list that Delphi can examine whenever it wishes to.

Installing Component Editors

Another nicety people do for custom component users is to install custom component editors, new items added onto a component's context menu whilst on the form designer. Listing 3 shows you that registered component editors are managed in just the same way as custom property editors, using a list of records.

Installing Experts

One way of installing something more dramatic than a component (or its support tools, the property editor and component editor) is to install an expert. For this, you inherit a class from the abstract `TIExpert` class, defined in the `ExptIntf` unit. To let Delphi know about it, you create an instance of the class and pass it to `RegisterLibraryExpert`. This goes

back to the component registration approach and uses an internal procedural variable, but this time the routine takes a reference to an actual object instance, rather than just a class reference.

Version 4 introduced a new Open Tools API unit, `ToolsAPI`, that allows you to write experts (now called wizards) using supplied interfaces. Listing 5 shows the parallel support for installing wizards using a reference to an interface implemented by an instance of a wizard object; again, not a class reference.

Package Registrations

The registration of property editors and component editors causes code in `DsgnIntf` (in the main VCL package) to add relevant structures to a list, also defined in this unit. The registration of component classes and expert objects involves calling a procedural variable, set up by the IDE (sometimes called a function pointer callback). With one approach (property/component editors) you can clearly see the implementation and maintenance code for the lists. However, Delphi adds a level of abstraction in the case of component classes and expert objects. Internally the same sort of approach is used: there is a list being maintained, regardless of where it is.

There is a reason for the difference in approach. The property/component editors are not used as soon as they are registered, but later on, when some object is selected on the form designer. On the other hand, as soon as component classes are registered, they are used by being installed on the component palette.

Similarly, when experts are registered, they are immediately installed into the IDE UI. In order to achieve this, the code that sets up the component classes and expert objects must explicitly manipulate the IDE. To avoid having all this IDE manipulation code in the VCL package, it is left in the main IDE application. The IDE simply sets a package function pointer variable to point to one of its own routines.

This is important to bear in mind when designing your own dynamic packages. If they are going to be registering things, you will need to consider whether the registered item will need to affect the main application immediately. If so, the function pointer callback approach is a good idea. If not, then leave the list manipulation code in the static package, the one required by the application.

```
type
  PPropertyClassRec = ^TPropertyClassRec;
  TPropertyClassRec = record
    Group: Integer;
    PropertyType: PTypeInfo;
    PropertyName: string;
    ComponentClass: TClass;
    EditorClass: TPropertyEditorClass;
  end;
var PropertyClassList: TList = nil;
procedure RegisterPropertyEditor(PropertyType: PTypeInfo; ComponentClass: TClass;
  const PropertyName: string; EditorClass: TPropertyEditorClass);
var P: PPropertyClassRec;
begin
  if PropertyClassList = nil then
    PropertyClassList := TList.Create;
  New(P);
  P.Group := CurrentGroup;
  P.PropertyType := PropertyType;
  P.ComponentClass := ComponentClass;
  P.PropertyName := '';
  if Assigned(ComponentClass) then
    P^.PropertyName := PropertyName;
  P.EditorClass := EditorClass;
  PropertyClassList.Insert(0, P);
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
type
  PComponentClassRec = ^TComponentClassRec;
  TComponentClassRec = record
    Group: Integer;
    ComponentClass: TComponentClass;
    EditorClass: TComponentEditorClass;
  end;
var ComponentClassList: TList = nil;
procedure RegisterComponentEditor(ComponentClass: TComponentClass;
  ComponentEditor: TComponentEditorClass);
var
  P: PComponentClassRec;
begin
  if ComponentClassList = nil then
    ComponentClassList := TList.Create;
  New(P);
  P.Group := CurrentGroup;
  P.ComponentClass := ComponentClass;
  P.EditorClass := ComponentEditor;
  ComponentClassList.Insert(0, P);
end;
```

Customising Tooltips

This was not really designed for extending the IDE through packages, but the principle can be used for exactly that purpose.

The `Controls` unit of the VCL (in VCL30.DPL or VCL40.BPL) defines a class `THintWindow` inherited from `TCustomControl`. This is a real, usable class, not an abstract class as in the case of experts. The reason this class is fully implemented is because it does a job, and it is used for the job it performs. As discussed in ***Hints With Attitude*** in Issue 16, `THintWindow` implements the hint window used for standard tooltips. However, as that article also explains, you can customise the hint windows, without the VCL knowing about it, thanks to the cunning way things have been organised by Borland.

As well as `THintWindow`, a class type, the `Controls` unit also defines `THintWindowClass`, a class reference type:

```
type
  THintWindowClass =
    class of THintWindow;
```

A variable defined of this type can be assigned `THintWindow` (yes, you can assign it a class) or any class inherited from `THintWindow`. The `Forms` unit declares a variable `HintWindowClass` of type:

```
var HintWindowClass:
  THintWindowClass = THintWindow;
```

Notice that by default it is initialised with THintWindow, but there is nothing stopping you replacing this with any custom class inherited from THintWindow. When the Application object needs a tooltip, it executes this statement:

```
FHintWindow :=
  HintWindowClass.Create(Self);
```

This calls the constructor of the appropriate class as referenced by HintWindowClass. Fortunately, all classes inherited from TComponent (including THintWindow) have polymorphic constructors. This fact means that no matter what class is referenced by the class reference variable, its own constructor will be executed (as opposed to THintWindow's constructor being called regardless, which would happen if the constructor were not polymorphic).

This business of having a polymorphic constructor seems to confuse a lot of people. To get all the low level details on why this is possible, and in fact necessary, for Delphi to work, have a flip through my *Fatal Startup Error* article in Issue 30. Also, if polymorphism is not your forté, you could take a look at the *Virtual and Override Clarification* entry in Issue 12's *Delphi Clinic*.

If you consider this idea, you could do much with it. If an application uses some custom object, then you can define a class reference type, and an initialised variable of that type whose constructor gets called. Make sure the constructor is polymorphic and then any installed packages can simply update the class reference variable to refer to a new class and the program will be given a new super-duper version of the expected object.

Arbitrary Add-In Packages

Another way of extending the IDE's functionality is to write an add-in package. Just as Delphi and your package share the same Classes unit (variables and all), the same is true of the Forms unit (and anything else in the VCL). The point is that Delphi's Application object is your own Application object. The MainForm property of your Application object is therefore the Delphi main window.

After absorbing this point, you tend to realise what can be done here. All you need to do is to write appropriate code that starts from the Application object, and you can tap into much of the IDE's make-up.

My article *Delphi 3 Add-In Packages: Digging The Dirt On Archaeopteryx* from Issue 27 discusses this concept in depth. We can apply exactly the same techniques to any other package-based application that can load up an arbitrary package.

Incidentally, if any Delphi 4 users are missing the Archaeopteryx functionality, and haven't already switched to the colossally more functional GExperts, I have finally got it working in Delphi 4 (and C++Builder 4). You can find the binary file in Dinosaur.zip on this month's disk, and the source in the file DinoSource.zip.

```
type
  TIExpert = class(TInterface)
  public
    ...
    function GetName: string; virtual; stdcall; abstract;
    function GetIDString: string; virtual; stdcall; abstract;
    function GetState: TExpertState; virtual; stdcall; abstract;
    procedure Execute; virtual; stdcall; abstract;
  end;
  TExpertRegisterProc = function(Expert: TIExpert): Boolean;
var LibraryExpertProc: TExpertRegisterProc = nil;
procedure RegisterLibraryExpert(Expert: TIExpert);
begin
  if @LibraryExpertProc <> nil then
    LibraryExpertProc(Expert);
end;
```

➤ Above: Listing 4

➤ Below: Listing 5

```
type
  IOTAWizard = interface(IOTANotifier)
    ['{B75C0CE0-EEA6-11D1-9504-00608CCBF153}']
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
  end;
  TWizardRegisterProc = function(const Wizard: IOTAWizard): Boolean;
var LibraryWizardProc: TWizardRegisterProc = nil;
procedure RegisterPackageWizard(const Wizard: IOTAWizard);
begin
  if Assigned(LibraryWizardProc) then
    LibraryWizardProc(Wizard);
end;
```

Forget The IDE... What About Our Application?

So far we have looked at a number of ways in which Delphi sets itself up to be customised by add-in packages. We will now need to turn our attention to how we can implement these ideas in our own applications, and what problems these may spring upon us.

First, we need to focus on what is required of an add-in package.

Requirements Of Dynamically Loaded Packages

The most important thing about a package is that it if it gets loaded explicitly by the application, it must also be explicitly unloaded by the application. This should happen either when the program is being closed down or at some point before, possibly due to user request. Because of this, you must keep track of all dynamically loaded packages to enable you to unload them. Some simple form of list should do the trick. The IDE uses lists to keep track of lots of things, including loaded packages, property editors, components editors, component classes and expert objects (or interfaces implemented by them).

The other primary point is that if any dynamically loaded package creates objects or registers classes, then these must be destroyed and unregistered respectively when the package is unloaded. This is of utmost importance as the code that implements

the methods of the objects and classes will vanish when the package is removed from memory. The same would be true if you set any function pointers in the program to point at code in a dynamic package. So when you unload a package you need to ensure that either the package cleans up after itself, or the program cleans up on behalf of the package.

There seem to be two common ways of attending to the timely departure of objects created by a dynamically loaded package. Let's see how they work.

Tracking Objects By Address

The first approach is exemplified in the aforementioned demo application on the Borland website. The demo is simple. The package has a form in it, and the form class is registered into the global class list with `RegisterClass`. The program loads the package, calls `GetClass` to translate a string version of the package form class name into a class reference, and then calls its constructor to produce the form. `RegisterClass` was examined in the aforementioned *Fatal Startup Error* article from Issue 30.

The form was created with the `Application` object as the owner. When the package needs to be unloaded, the program iterates through the `Application` object's `Components` array (all the forms and components owned by it), looking for a form whose definition lives in the package. In an attempt to be generic, the program doesn't compare the form's class name with the previously used string. Instead, for each form, it uses the `ClassType` method to get a class reference to the class type. A class reference is actually a pointer to the class VMT, and a few other bits and pieces, all of which will live in the package address space, if this is the package's form.

The code then performs some Windows API magic to see if the base address of the memory block occupied by what the class reference points to is the same as the base address of the memory occupied by the package (Listing 6). If it is, the form comes from the package, and so is freed. When the loop through `Application.Components` is done, the package is unloaded.

The code certainly seems to work, but I think it could be slightly shorter. It calls `ClassName` from a class, to get a string containing the class name. Then it passes that back to `GetClass`, to ensure that a class reference is returned that has been previously registered. I suspect you get the same effect by simply calling `ClassType`, to return a class reference directly. Granted there is no assurance that the class is registered, but that's not the important point. The key thing is that the class reference contains the address that may be found to match that of the package.

This approach of using `VirtualQuery` is also used by a routine in the `Classes` unit. `UnRegisterModuleClasses` (called in Listing 6) loops through the registered classes, removing those that were registered by the specified package.

Tracking Objects By Group

An alternative way to deal with a number of items manufactured by a particular package is to do what the IDE does with items installed through the Open Tools API. Take another look at Listings 2 and 3. Pay particular attention to the records copied from the `DsgnIntf` unit, used to keep track of the property and component editors.

Notice they both have a `Group` field, declared as an `Integer`. Also notice that when this field gets initialised by the relevant registration routine, in both cases the value comes from something called `CurrentGroup`. This is a non-local variable declared in the `Classes` unit, with an initial value of -1. A comment nearby describes this variable as holding the *current design group*.

When the IDE is asked to load up a package (or is about to load a package it remembers from the previous session), it calls the `NewEditorGroup`, as defined in the `DsgnIntf` unit. The idea is that each package is given a unique number, called a design group or editor group.

Whenever a package calls a registration routine, such as those described earlier in this article, the registration routine adds the group number (to identify which package it came from) into the manufactured registration record.

The key point here is that when a package needs to be unloaded, Delphi can loop through all these lists holding the registration records describing things registered by packages. When it comes across something with a group number that corresponds to the package in question, any tracked resources can be deleted safely, and those registration records can be removed from the list.

The `FreeEditorGroup` procedure in `DsgnIntf` does just this. It loops through the property editor, component editor and property mapper list, tidying up anything that belongs to the passed in design group.

As packages are unloaded, group numbers are freed up, and can be used again by any packages which are loaded later. `NewEditorGroup` uses a `TBits` object to maintain these group numbers (a private variable called `EditorGroupList` in the `DsgnIntf` unit).

In your own applications, you can make use of `NewEditorGroup` and `CurrentGroup`, although you

➤ *Listing 6*

```
procedure UnLoadAddInPackage(Module: THandle);
var
  I: Integer;
  M: TMemoryBasicInformation;
begin
  { Make sure there are no instances of any classes from Module instantiated, if
    so free them. This assumes that the classes are owned by the application. }
  for I := Application.ComponentCount - 1 downto 0 do begin
    VirtualQuery(GetClass(Application.Components[I].ClassName), M, SizeOf(M));
    if (Module = 0) or (HModule(M.AllocationBase) = Module) then
      Application.Components[I].Free;
  end;
  UnRegisterModuleClasses(Module);
  UnloadPackage(Module);
end;
```

will have to re-implement FreeEditorGroup to loop through your own package-related lists. Given that EditorGroupList is not available from the DsgnIntf unit, you may also need to re-implement NewEditorGroup (or an equivalent) to ensure efficient use of the freed up bits in the TBits object, if that is what you choose to use.

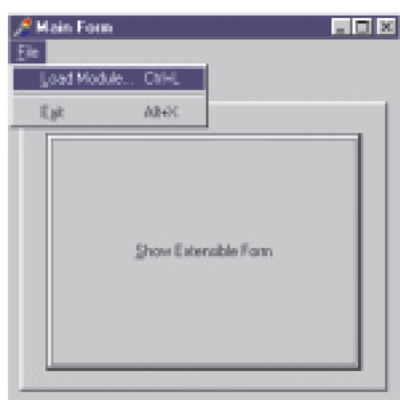
Dave Jewell briefly mentioned this CurrentGroup variable in his column in Issue 30, *Beating The System: Exploring Delphi's Closed-Tools API, Part 2*. He also referred to an internal IDE method, UnregisterExpertGroup, that unloads all experts related to a given editor group, which is much the same as what we see happening for property/component editors in the FreeEditorGroup procedure.

The App That Jack Built

Now all the groundwork has been completed, and we can consider ourselves jacks of all (package-related) trades, we can progress onto a sample package-based distributed dynamic application. The finished application that we are working towards is on the disk, as per usual, but I should mention something about the files.

Because of certain differences in package management by the Delphi 3 and Delphi 4 IDEs, I have supplied two sets of files, one for each version. The unit files are supplied once only, but each project and package file is supplied twice. If it has a 3 at the end of its name, then it is for Delphi 3. If it has a 4 at the end, then it is for Delphi 4 (or possibly later). I'll be referring to the project/package names with the 4 at the end, but unless noted otherwise everything applies equally to the Delphi 3 versions.

> *Figure 2: The sample application dynamically loads packages as well.*



In order to distinguish packages designed for this particular application from normal Delphi packages, or those for other applications, I have decided to prefix them with BL, so a file mask of BL*.BPL (or BL*.DPL for Delphi 3 packages) will locate them.

The main application on the disk is called AppProject4.dpr. The main form is called (unimaginatively) MainForm, and its form unit is called AppMainFormU.pas. The user interface is sparse: there is a menu and a button on the single tab sheet of a page control; the menu has an exit option (which calls Application.Terminate) and a Load Module... option (see Figure 2).

Loading Packages

This Load Module... menu item launches another form from the application (ModuleLoadForm in unit AppModuleLoadFormU) whose job is to locate extra application modules and load them.

Of course the program could have dispensed with this form and simply searched for appropriate files to load. In this case, I went for the manual load approach.

The form attempts to be reasonably similar to the Delphi package loading form (Figure 1 again) in that it has Add... and Remove buttons, and a listbox that will show which packages are currently loaded. The listbox shows the package descriptions and, when one is selected, its file name is shown (trimmed to fit under the listbox if necessary).

```
procedure TModuleLoadForm.BtnAddPackageClick(Sender: TObject);
begin
  if DlgOpenPackage.Execute then begin
    LoadCustomPackage(DlgOpenPackage.FileName);
    FormatPackagesAsDisplayList(LstPackages.Items)
  end
end;
procedure TModuleLoadForm.BtnRemovePackageClick(Sender: TObject);
begin
  if LstPackages.ItemIndex <> -1 then begin
    UnloadCustomPackage(
      HModule(LstPackages.Items.Objects[LstPackages.ItemIndex]));
    FormatPackagesAsDisplayList(LstPackages.Items)
  end
end;
procedure TModuleLoadForm.LstPackagesClick(Sender: TObject);
begin
  //FileCtrl.MinimizeName inserts dots to shorten
  //a long path name for display purposes
  LblPackageFileName.Caption := MinimizeName(
    TPackageRec(PackageList[LstPackages.ItemIndex]^).FileName,
    Canvas, LstPackages.Width)
end;
```

> *Listing 7*

The code in the event handlers of these buttons and the listbox is simple enough, it involves calls to other more involved routines in the AppSupportU helper unit, as Listing 7 shows. The listbox event handler will make more sense after reading the following paragraph.

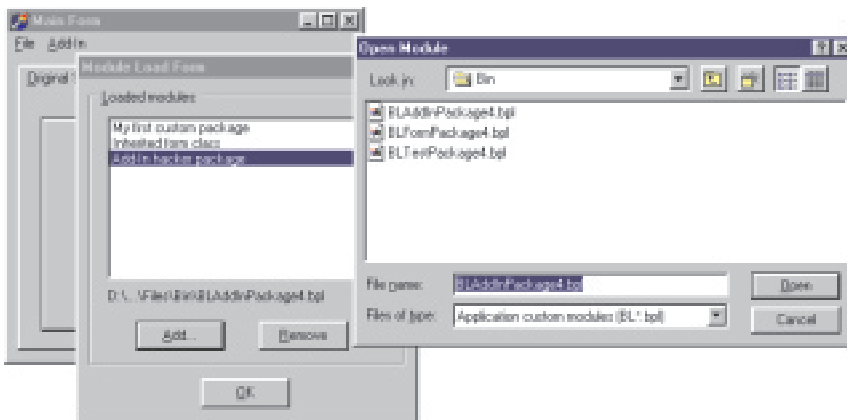
The LoadCustomPackage routine makes a call to LoadPackage (from Sysutils). LoadPackage returns a HModule used to identify the package. Once the package has been loaded, a record describing the package is set up and added to a TList object called PackageList. Listing 8 shows all the code relating to this routine. You can also see that a check is made to verify that the requested package has not already been loaded, before committing to loading it.

This could equally be managed using a simple object to represent the package, with fields for the module, description etc. The object destructor could ensure the package gets unloaded.

When the package is successfully loaded, a new group number is set up, to facilitate easy tracking of installed items that came from this package. Two routines, NewAppGroup and FreeAppGroup, see to this, working with a TBits object called AppGroupList to efficiently manage group numbers, ensuring freed up group numbers are re-used. FreeAppGroup will be where we eventually ensure all

```
type
  PPackageRec = ^TPackageRec;
  TPackageRec = record
    FileName,
    Description: String;
    Module: HModule;
    Group: Integer;
  end;
  EPackageLoadError = class(EPackageError);
var
  PackageList: TList;
  AppGroupList: TBits = nil;
function NewAppGroup: Integer;
begin
  if AppGroupList = nil then
    AppGroupList := TBits.Create;
  CurrentGroup := AppGroupList.OpenBit;
  AppGroupList[CurrentGroup] := True;
  Result := CurrentGroup;
end;
procedure FreeAppGroup(Group: Integer);
begin
  //Destroy any objects that were created by this group
  //...(to be implemented)...
  if (Group >= 0) and (Group < AppGroupList.Size) then
    AppGroupList[Group] := False;
end;
// Simple wrapper for SysUtils.LoadPackage which also adds
// to the package list
function LoadCustomPackage(const Name: String): HModule;
var
  P: PPackageRec;
  Loop: Integer;
begin
  for Loop := 0 to PackageList.Count - 1 do
    with TPackageRec(PackageList[Loop]^) do
      if AnsiCompareFileName(Name, FileName) = 0 then
        raise EPackageLoadError.CreateFmt(
          'Package already loaded:'#13'  %s'#13'  %s',
          [FileName, Description]);
  Result := LoadPackage(Name);
  New(P);
  P.Module := Result;
  P.FileName := Name;
  P.Description := GetPackageDescription(PChar(Name));
  CurrentGroup := NewAppGroup;
  P.Group := CurrentGroup;
  PackageList.Add(P)
end;
procedure InitializePackageSupport;
begin
  PackageList := TList.Create;   //Create package list
  RegisterPagesProc := InternalRegisterPages
end;
procedure FinalizePackageSupport;
begin
  while PackageList.Count > 0 do   //Unload packages
    UnloadCustomPackage(PPackageRec(PackageList[0]).Module);
  PackageList.Free;   //Delete package list
  AppGroupList.Free;
  PageFormList.Free   //Free page control form list
end;
```

➤ *Listing 8*



➤ *Figure 3: The application keeps a list of all loaded packages.*

package-manufactured objects are disposed of sensibly.

The package description is extracted by the `SysUtils` routine `GetPackageDescription`. This is used to display in the module load form listbox in Figure 3, rather like Delphi does in Figure 1.

Unloading Packages

Looking back at Listing 7, the other mentioned routines, `UnloadCustomPackage` and `FormatPackagesAsDisplayList`, are shown in Listing 9. `UnloadCustomPackage` undoes the work of `LoadCustomPackage`, calling `FreeAppGroup`, unloading the package from memory and then deleting the package description record from `PackageList`. The `AppSupportU` unit's finalisation routine (called as the main form is closing down) in Listing 8 will unload dynamically loaded packages that are still hanging around using calls to `UnloadCustomPackage`.

The routine `FormatPackagesAsDisplayList` fills a passed-in `TStrings` object with package descriptions for all loaded packages. This is used by the module load form to fill up the listbox after loading or unloading a package. As well as just storing the package description as the string, the module handle is also stored in the `Objects` array. This allows the form to successfully call `UnloadCustomPackage` when the `Remove` button is pressed, the routine takes a module handle.

The module load form also ensures that, if compiled under Delphi 3, the open dialog uses .DPL extensions for package files instead of .BPL extensions, using conditionally compiled code in the form's `OnCreate` event handler.

Keeping Track Of Packages

Another aspect of the AppProject4 application is that it remembers which packages have been loaded. When the main form closes, details of all the packages that are still open are written in the registry before they get unloaded by the `AppSupportU` finalisation routine. Also, when the main form starts up, it reads all the known packages from the same registry section and loads them, after package support has been initialised. Listing 10 has the main form event handlers, and the support routines they call.

This is exactly what Delphi does. Both Delphi 3 and 4 have a `Known Packages` section in the registry, listing add-in packages that implement components etc. Delphi 4 also has a `Known IDE Packages` section, for packages implementing portions of the IDE itself.

Calling Package Registration Routines

As I mentioned before, when Delphi loads a package, it iterates through the units in the package, looking for `Register` procedures, calling them as it finds them. You might have cause to wonder whether you can also locate an arbitrary routine in your dynamically loaded packages, and execute it. If you are used to playing with DLLs, you may consider that a

package is a special form of DLL, and be tempted to pass a target routine name to `GetProcAddress`, but this in itself would not be enough. There could be many routines of the same name made available from a number of units contained in the package, and they wouldn't all be exported with the same name.

Before examining how to successfully access a routine in some arbitrary dynamically loaded package, we should really look at the issue of whether this should be done at all. Yes, it is a fact that Delphi calls the `Register` routine from any units contained in the package, but this was really a forced decision, for backwards compatibility. The Delphi 1 and 2 component libraries used the `Register` routine as the mechanism for component authors to install their wares. Now that Delphi employs packages, old component source still needs to work, so Delphi must locate these `Register` routines and call them.

However, the intention of the Delphi package mechanism is a way of partitioning one application into manageable executable portions without affecting the semantics of the program in any way. 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 source code or organisation.

You should not concern yourself with calling from the EXE to the package DLL, you just call something in a unit and Delphi sorts out appropriate code to cross module boundaries where necessary. Implementing packages as DLLs was convenient for Borland to do so they could take advantage of the Windows DLL loader to do the runtime linking. Implementing their own linking using compiler-generated code would have no doubt been very complex and very difficult to ensure reliability with.

Borland R&D advise that if you want DLL operations (like locating a routine in a package DLL and calling it), then you should use DLLs. The Delphi IDE only accesses `Register` in this DLL-like fashion. Every other form of inter-package communication is done through unit initialisation sections and function pointer callbacks, such as `RegisterComponents` and `RegisterLibraryExpert`.

Their suggestion is to do as Delphi mostly does. Define a registration function in package A, required to be used by all packages you want to dynamically load. In dynamically loaded package B, call the registration function. B will be statically linked to A, due to the package requirement clause, so this is a simple function call set up by the linker. In the case of Delphi and the previously mentioned function pointer callbacks, package A is VCL30.DPL or VCL40.BPL, where the registration routine is defined in `DsgnIntf` or `ExptIntf`, and package B is any design-time package.

In many cases, if the requirement is for a registration routine to be called, you can use a unit initialisation section instead. Unlike units contained in an implicitly linked package, all units contained in a dynamically loaded package will be initialised regardless of

```
{ Code to take package list and extract a displayable subset. Target TStrings
  object has descriptions added, as well as module handles (in Objects array) }
procedure FormatPackagesAsDisplayList(List: TStrings);
var Loop: Integer;
begin
  List.BeginUpdate;
  try
    List.Clear;
    for Loop := 0 to PackageList.Count - 1 do
      with TPackageRec(PackageList[Loop]^) do
        List.AddObject(Description, TObject(Module))
  finally
    List.EndUpdate
  end
end;
{ Simple wrapper for SysUtils.UnloadPackage which also removes from package list}
procedure UnloadCustomPackage(PackageModule: HModule);
var Loop: Integer;
begin
  for Loop := 0 to PackageList.Count do
    if PPackageRec(PackageList[Loop]).Module = PackageModule then begin
      FreeAppGroup(PPackageRec(PackageList[Loop]).Group);
      UnloadPackage(PackageModule);
      Dispose(PackageList[Loop]);
      PackageList.Delete(Loop);
      Break
    end
end;
```

➤ *Above: Listing 9*

➤ *Below: Listing 10*

```
procedure TMainForm.FormShow(Sender: TObject);
begin
  InitializePackageSupport;
  LoadPackagesStoredInRegistry
end;
procedure TMainForm.FormHide(Sender: TObject);
begin
  StorePackagesInRegistry;
  FinalizePackageSupport
end;
const
  RegPath = 'Software\Oblong\AppProject';
  RegSection = 'Known Modules';
procedure LoadPackagesStoredInRegistry;
var
  Pkgs: TStrings;
  Loop: Integer;
begin
  with TRegIniFile.Create(RegPath) do
    try
      Pkgs := TStringList.Create;
      try
        ReadSection(RegSection, Pkgs);
        for Loop := 0 to Pkgs.Count - 1 do
          LoadCustomPackage(Pkgs[Loop])
      finally
        Pkgs.Free
      end
    finally
      Free
    end
end;
procedure StorePackagesInRegistry;
var
  Loop: Integer;
begin
  with TRegIniFile.Create(RegPath) do
    try
      EraseSection(RegSection);
      for Loop := 0 to PackageList.Count - 1 do
        with TPackageRec(PackageList[Loop]^) do
          WriteString(RegSection, FileName, Description)
    finally
      Free
    end;
end;
```

whether or not anything defined within them is accessed.

However, there are downsides to this. There may be a possibility that other units have not executed their own initialisation sections when yours does, meaning some things may not be set up as you like. Also, the code will always execute whether you want it to or not and you cannot re-execute this code without unloading and reloading the package.

Accessing One Routine In A Package

So let's say you decide that you really do want to locate some routine in a unit of your package, or even a routine in *all* the units in your package. What is the correct approach? We'll try the easier one first, one specific routine in one specific unit in the package. Unfortunately the approach differs for Delphi 3 and 4.

Delphi packages export symbols to allow access to all routines, methods, classes, and variables. Since there may be symbols defined with the same name in more than one unit, the exported symbol name includes the identifier name and the unit name. Delphi 3 manufactures the export name using the following pseudo-call to `Format`, where `HashValue` is the compiler's internal symbol version hash value:

```
Format('%s.%s@%.8x', [UnitName,
  IdentifierName, HashValue])
```

It seems to be impossible to pre-calculate this hash number, so the easiest thing is to add the target function into a package, compile it to a `.DPL` file and then run `TDump` across it, looking for the list of exports. In the case of a `Register` routine, you will find it encoded as:

```
Format('%s.Register@51F89FF7',
  [UnitName])
```

Delphi 4 takes a different approach. Because of the requirement to have package compatibility with C++Builder 4, Inprise chose to use C++ name-mangling on the symbols (but also to include the unit name in the mangled result). Name mangling is a process where the routine's signature is textually encoded to create a unique symbol name for the linker and export list. This includes all the parameter types, the return type and also the calling convention.

Whilst this means that we can rely on C++Builder 4 and Delphi 4 generating the same export name for a given routine, the name mangling stuff is compiler version dependent and subject to change on a whim, so you can readily fall foul of any future changes.

With version 4, a routine called `Register` with no parameters, using the default `register` calling convention (`fastcall` for C programmers), is exported using this pseudo-call to generate the string:

```
Format('@%s@Register$qqrv',
  [UnitName])
```

Again, we can use `TDump` to help us out here, but be warned. `TDump` will un-mangle the exported package routines by default, showing all the parameter types and calling conventions (in C syntax), unless you use the `-m` command-line switch.

So now we can pass this generated name to `GetProcAddress`, locate the routine and call it.

Calling A Routine In All Units Of A Package

So now we know how to get at a routine, how do we know all the units compiled into a package? For the answer, we can again check out the `SysUtils` unit. This implements another useful package-related function called `GetPackageInfo`. The Delphi help documents this as a routine that enumerates over all the contained units (and required packages) in the package, calling a supplied callback routine for each one. Dave Jewell mentions the routine in his column in Issue 27, *Beating the System: EXE Sniffing, The Story Continues*.

Listing 11 shows the callback routine, and also the call to `GetPackageInfo` that has been inserted into the `LoadCustomPackage` function (originally from Listing 8). If you study the listing, you can see that in versions of Delphi later than 3, when a symbol name is mangled, it always formats the unit name with an initial capital and the rest in lower case. These export names are case-sensitive, so we need to follow this rule.

You will also see that in my sample packages, the registration routine is called `BLRegister`. Since exports are indeed case-sensitive, any package that implements such a registration/initialisation routine must use the same capitalisation. The same point is made about the Delphi registration routine, `Register`, in the *Delphi 4 Components* entry of Issue 42's *Delphi Clinic*.

Three of the sample packages supplied have a `BLRegister` routine. The first package BLTestPackage4.bpl uses it to display a message box. The other ones, BLPageFormPackage4.bpl and BLPageFormPackage4.bpl, use `BLRegister` to actually install their additional functionality.

Writing An Extender Package

So here we have a new term that I've just made up ☺. An *extender package* is one that is intended to be dynamically loaded for the purposes of extending the functionality of an application.

The application that goes with this article comes with a few extender packages. The source code is supplied (of course) and I have also supplied compiled versions of these packages (and the application) for Delphi 3 and 4. These binary files are quite small thanks to them all being compiled with package support.

My First Extender Package

The first package to test the waters with is very primitive. The BLTestPackage4.bpl (as previously mentioned) contains one simple unit. To prove the principle, the initialisation and finalisation sections display a simple message box to indicate they are executing. Also, the unit has a `BLRegister` routine which also displays a simple message box.

There's not much more to say about this package, apart from the fact that it can indeed be loaded and unloaded at will by the application, with the message boxes popping up at the appropriate times. With the concept working, let's try some real extender packages.

Class References And Inheritance

The first real extender package we will look at will implement an example of the idea described above in the section *Customising Tooltips*. As mentioned, a variable is used by the application, but which is defined using a class reference type, instead of a specific class type. In terms of the 'package A, package B' notation used earlier, package A will define a usable class and a class reference type, plus an appropriate variable. The application requires package A. Package B also requires package A, and implements a descendant of the base class defined in package A.

When package B is dynamically loaded, it overwrites the class reference variable in package A with the class defined in package B. When the application creates an instance of the class using the class reference variable, it will in fact create an instance of package B's class, something it was not compiled with. In the example in hand, package A is called CorePackage4.bpl and the add-in package B is called BLInheritedFormPackage4.bpl. The classes used for this example are actually form classes, the example shows how form inheritance can be used by an extender package to enhance the functionality of an application.

CorePackage4.bpl defines a form `BaseForm` of type `TBaseForm` in the `BaseFormU` unit. It also defines a class reference type `TBaseFormClass` in terms of `TBaseForm` and defines a `TBaseFormClass` variable, `BaseFormClass`, initialised to `TBaseForm`. When the BLInheritedFormPackage4.bpl package gets loaded, the `initialisation` section of its `InheritedFormU` unit (which defines a `TInheritedForm` class, inherited from `TBaseForm`) sets the `BaseFormClass` variable to `TInheritedForm`.

The button on the application's main form (see Figure 2) creates a form through the class reference variable:

```
with BaseFormClass.Create(
  Application) do
  try
    ShowModal
  finally
    Free
  end
```

In this case, the application already had some functionality, which was extended by the package. Figure 4 shows what happens when the button is pushed without the new package loaded and Figure 5 shows the effect of the button after having loaded BLInheritedFormPackage4.bpl.

When the package is unloaded the `finalisation` section of `InheritedFormU` resets the `BaseFormClass` variable back to `TBaseForm`.

Abstract Classes And Polymorphism

In the case above, the original form class `TBaseForm` was a normal, usable class. It was actually used by the original application (as per Figure 4). You can make the class much more extendible by defining appropriate virtual methods.

In fact, you could take this whole idea to a much more abstract level. Your application's main required package (package A) can define completely abstract classes or interface types. Dynamically loaded packages can define classes inherited from the abstract base classes, or classes which implement those interfaces, and register them. The application, having been told to register these new classes, can now use this new functionality.

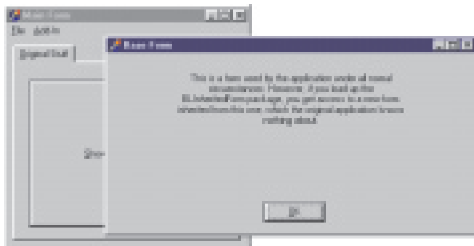
Your application only deals with the common abstract base class. Assuming you put enough thought into the class, to make it extensible, dynamic packages can do a lot to extend the application. OOP is a very powerful technique here, and is exactly that used by Delphi when dealing with experts. It is only aware of the abstract class `TIExpert`, or the `IOTAWizard` interface.

Registering Items From A Package

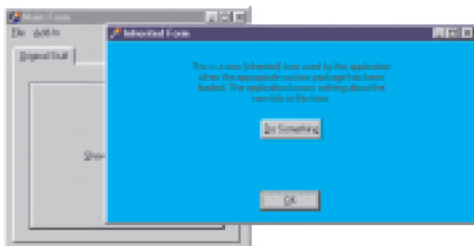
The next extender package creates a form object, then passes it to a registration routine in the application's CorePackage4.bpl package. The idea is that the form will be used as a new page on the page control (which defaults to having only one page, and is partially

➤ Listing 11

```
procedure PackageInfoProc(const Name: string;
  NameType: TNameType; Flags: Byte; Param: Pointer);
type
  TRegisterProc = procedure;
var
  RegisterProc: TRegisterProc;
  UnitName, ProcName: String;
const
{$ifdef Ver100} //Delphi 3
  ExportName = '%s.BLRegister@51F89FF7';
{$else}
  ExportName = '@%s@BLRegister$qqrv';
{$endif}
begin
  if NameType = ntContainsUnit then begin
  {$ifdef Ver100} //Delphi 3
    //Delphi 3 packages don't use name-mangling
    //Unit names maintain their original case
    UnitName := Name;
  {$else}
    //Delphi 4+ mangles names - the unit name is all
    //lower case, with an initial capital letter
    UnitName := LowerCase(Name);
    if Length(UnitName) > 0 then
      UnitName[1] := UpCase(UnitName[1]);
  {$endif}
    ProcName := Format(ExportName, [UnitName]);
    @RegisterProc := GetProcAddress(PPackageRec(
      Param).Module, PChar(ProcName));
    if Assigned(RegisterProc) then
      try
        RegisterProc
      except
        on E: Exception do
          ShowMessageFmt('Error %s registering %s package',
            [E.ClassName, PPackageRec(Param).FileName])
      end
  end
end;
function LoadCustomPackage(const Name: String): HModule;
begin
  ...
  GetPackageInfo(P.Module, P, PackageFlags,
    PackageInfoProc);
end;
```



➤ *Figure 4: A form as implemented in the program.*



➤ *Figure 5: A new inherited form, dynamically loaded.*

visible in Figures 2, 3, 4 and 5). To make a form act as the client area of a tab sheet in a page control relies upon setting a number of its properties.

Because this form object is to be immediately added to the page control, as discussed earlier in *Package Registrations*, the registration routine is implemented via a function pointer callback. The CorePackage4.bpl package's `CommonHookU` unit defines a routine much like Delphi's `RegisterLibraryExpert` routine. Assuming a pointer (`RegisterPagesProc`) has been set up with a value, `RegisterPages` will call it, passing a desired tab sheet caption and the form object, along with the current package group (Listing 12).

The application has the real registration routine in the `AppSupportU` unit, called `InternalRegisterPages`. It is implemented in much of a similar way to Delphi's `RegisterComponentEditor` and `RegisterPropertyEditor`. It uses a list (`PageFormList`) of records of type `TPageFormRec` to maintain the form objects and their associated package group.

Now that a package is actually manufacturing objects, we need to ensure they get tidied up correctly. This means that the `FreeAppGroup` procedure from Listing 8 needs to be extended to achieve this requirement. The extra code loops through `PageFormList`, checking the `Group` field of the `TPageFormRec` record. If it finds a record from the specified package group, it frees the page form, removes the record from the list and frees the record from memory.

Of course, if we are managing objects through interfaces, we simply need to assign `nil` to the interface reference, and the object will make sure it destroys itself.

This tidying up code is in the `UnloadPages` procedure, shown with `InternalRegisterPages` in Listing 13.

This package, BLPageFormPackage4.bpl, registers this new page form object in its `BLRegister` routine using this statement:

```
RegisterPages(
  'Windows &version',
  [TNewPageForm.Create(nil)])
```

The specifics of the actual form implemented by this extender package are irrelevant, strictly speaking, in the context of this article. However, I will mention that the form is a very simple aqua-coloured form, which displays some standard Windows version information stored in variables in the `SysUtils` unit. Figure 6 shows the new form in situ in the page control, along with the module load form highlighting this particular package, and showing its file name.

Non-Specific Add-In Packages

The final package to look at is BLAddInPackage4.bpl. This package takes the approach of not using any formal installation or registration routine to install itself. Instead, it acts in a more nefarious manner. As described in the earlier section *Arbitrary Add-In Packages*, this package locates the `Application` object of the application, then uses the `MainForm` property to find the main form. It then continues locating objects until it finds what it is looking for, and manipulates those components how it likes, primarily to insert new objects into the user interface.

In this case, it adds a new menu into the main form's main menu to allow additional behaviour to be accessed by the user. All the necessary techniques for this are explained in the article on writing packages like this for Delphi's IDE, from Issue 27, so I won't hark on about them here. Instead, I'll refer you back to that article.

As it happens, the new menu item's job is to invoke a form that also comes from the add-in package. This form is an object browsing form (see Figure 7). The idea is to help you identify the layout of the application you are writing an add-in for.

When the form is visible, the component ownership hierarchy is shown at the top left of the form. The `Application` object is at the top of this hierarchy, followed by all components it owns, and so on. When you select a component, the other portions of the form come into play. The bottom right tree view shows the component's inheritance hierarchy, all the way up to `TObject`. The grid on the right of the form is a mock-up of the Object Inspector and shows the values of the component's published properties. If the selected component is a visual component that is a parent of other components, the bottom left hierarchy shows the parent/child relationships.

I knocked up this form in front of the TV one evening, as something to do. However, from this understated beginning I have found it to

➤ *Listing 12*

```
var
  RegisterPagesProc: procedure(Group: Integer; const Page: String;
  PageForms: array of TForm) = nil;
procedure RegisterPages(const Page: String; PageForms: array of TForm);
begin
  if Assigned(RegisterPagesProc) then
    RegisterPagesProc(CurrentGroup, Page, PageForms);
end;
```

```
type
  PPageFormRec = ^TPageFormRec;
  TPageFormRec = record
    Group: Integer;
    PageForm: TForm;
    TabSheet: TTabSheet;
  end;
var PageFormList: TList = nil;
procedure InternalRegisterPages(Group: Integer; const Page:
  String; PageForms: array of TForm);
var Loop: Integer;
begin
  for Loop := Low(PageForms) to High(PageForms) do
    LoadPage(Group, Page, PageForms[Loop])
end;
procedure LoadPage(Group: Integer; const Page: String;
  PageForm: TForm);
var P: PPageFormRec;
begin
  if PageFormList = nil then
    PageFormList := TList.Create;
  New(P);
  P.Group := CurrentGroup;
  P.PageForm := PageForm;
  P.TabSheet := TTabSheet.Create(nil);
  PageFormList.Insert(0, P);
  P.TabSheet.Parent := MainForm.PageControl;
  P.TabSheet.PageControl := MainForm.PageControl;
  P.TabSheet.Caption := Page;
  MainForm.PageControl.ActivePage := P.TabSheet;
  with P.PageForm do begin
    Hide;
    Left := 0;
    Top := 0;
    BorderStyle := bsNone;
    Parent := P.TabSheet;
    WindowState := wsMaximized;
    Show
  end;
end;
procedure UnloadPages(Group: Integer);
var
  I: Integer;
  P: PPageFormRec;
  PageCtl: TPageControl;
begin
  if not Assigned(PageFormList) then Exit;
  I := PageFormList.Count - 1;
  while I > -1 do begin
    P := PageFormList[I];
    if P.Group = Group then begin
      PageCtl := P.TabSheet.PageControl;
      //Switch to a page that we are not removing
      if Assigned(PageCtl) and
        (PageCtl.ActivePage = P.TabSheet) then
        PageCtl.SelectNextPage(False);
      P.PageForm.Free;
      P.TabSheet.Free;
      PageFormList.Delete(I);
      Dispose(P);
    end;
    Dec(I);
  end;
end;
```

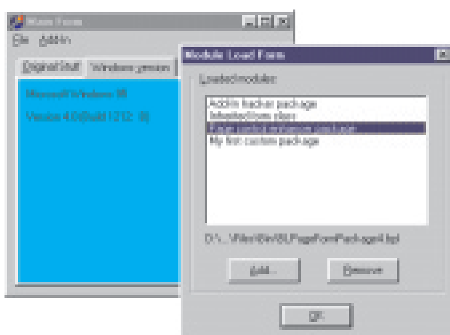
➤ *Listing 13*

be an extremely useful tool (in this arena of dynamically loaded packages). As a consequence I threw it into this article's sample project to allow you to also take advantage of it. As mentioned, the add-in package makes the form available from a new menu item on the application's main form. It installs itself from within the `BLRegister` routine (in the `AddInU` unit).

To make the package even more useful, I added another unit, `AddInIDEU`. This implements an IDE expert which is registered from the normal Delphi `Register` routine. So you can load this package both into the sample application that goes with this article, and also into Delphi itself. Figure 8 shows the form after having been installed into Delphi, looking at Delphi's main form (called `AppBuilder`).

Final Note

One last thing... If you load up the sample projects and packages and compile them, you must check all the search paths to ensure Delphi can always find the right files. Also, be sure to compile CorePackage4.bpl before compiling the application (which requires it). Of course, you also need to ensure that any application you make using this dynamic package loading technique is itself compiled with package support enabled.



➤ *Figure 6: Custom pages installed dynamically.*

Summary

The primary objective of this rather lengthy article was to give some ideas on how to write extensible applications using the Delphi 3 (and higher) package mechanism along the same lines as the Delphi IDE itself. Using abstract classes and initialised non-local class reference variables, you can provide standard back doors into the application. But a package-based application designed for extensibility can also be tweaked and tinkered with in any number of ways by handy (or indeed potentially harmful) additional modules.

If I've done a good job, you should be able to see that this approach could provide a potentially very powerful mechanism for supplying your customers with an application which can vary in its offered functionality. Customers could purchase modules as they require them. Additionally, you can supply customers with initial functionality early on in a product development cycle, which can then be added to as development continues, by supplying more and more packages that implement the remaining functionality.

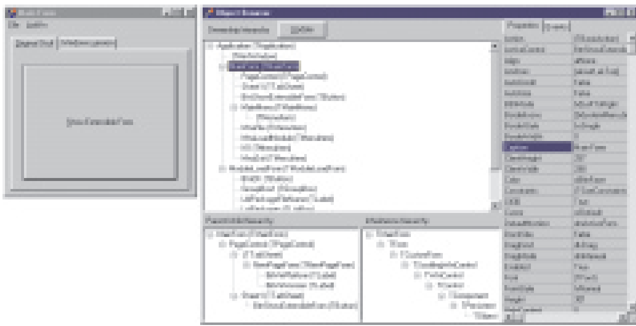
For extra reading, check out the numerous references that have been made throughout the text to other articles in back issues of *The Delphi Magazine*. Alternatively, just look up an appropriate topic, such as *property editors* on *The Delphi Magazine Collection CD*.

Acknowledgements

Some of the information and advice used in this article is based on comments made by Danny Thorpe and Allen Bauer, both Senior Inprise R&D engineers.

Brian Long is an independent consultant and trainer. You can reach him at brian@blong.com

Copyright @ 1999 Brian Long. All rights reserved.



➤ *Above: Figure 7: Add-in package's browser form.*

➤ *Below: Figure 8: Browsing the Delphi IDE.*