Packages to the Max

CHAPTER

14

IN THIS CHAPTER

Delphi 3 introduced *packages*, which enable you to place portions of your application into separate modules that can be shared across multiple applications. Packages are simply special *dynamic link libraries (DLLs)* that contain additional Delphi specific information. They differ from DLLs in how they are used. Packages are primarily used to store collections of components in a separate, sharable module (a Borland Package Library, or `.bpl` file). As you or other developers create Delphi applications, the packages you create can be used by the application at runtime instead of being directly linked at compile/link time. Because the code for these units resides in the `.bpl` file rather than in your `.exe` or `.dll`, the size of your `.exe` or `.dll` can become very small.

Packages are specific to the VCL; that is, applications written in other languages can't use packages created by Delphi (with the exception of C++Builder). One of the reasons behind packages was to get around a limitation of Delphi 1 and 2. In these prior versions of Delphi, the VCL added a minimum of 150KB to 200KB of code to every executable. Therefore, even if you were to separate a piece of your application into a DLL, both the DLL and the application would contain redundant code. This was especially a problem if you were providing a suite of applications on one machine. Packages allow you to reduce the footprint of your applications and provide a convenient way for you to distribute your component collections.

Why Use Packages?

There are several reasons why you might want to use packages. Three important reasons are discussed in the following sections: code reduction, application partitioning, and component containment.

Code Reduction

A primary reason behind using packages is to reduce the size of your applications and DLLs. Delphi already ships with several predefined packages that break up the VCL into logical groupings. In fact, you can choose to compile your application so that it assumes the existence of many of these Delphi packages.

A Smaller Distribution of Applications—Application Partitioning

You'll find that many programs are available over the Internet as full-blown applications, downloadable demos, or updates to existing applications. Consider the benefit of giving users the option of downloading smaller versions of the application when pieces of the application might already exist on their system, such as when they have a prior installation.

By partitioning your applications using packages, you also allow your users to obtain updates to only those parts of the application that they need. Note, however, that there are some versioning issues that you'll have to take into account. We'll cover these versioning issues in this chapter.

Component Containment

Probably one of the most common reasons for using packages is the distribution of third-party components. If you are a component vendor, you must know how to create packages because certain design-time elements—such as component and property editors, wizards, and experts—are all provided by packages.

Packages Versus DLLs

Using DLLs to host administrative forms for their server applications results in the DLL having its own copy of `Forms.pas`. This will cause a weird error involving Windows' handling of the window handles generated within the DLL—when the DLL is unloaded, the Window handle isn't dereferenced by the operating system. The next message that crosses the queue for all top-level windows causes a fault at the application, which the operating system then shuts down because the application is in an invalid state. Using packages instead of DLLs overcomes this problem because the packages refer to the main application's copy of `Forms.pas`, and the message queue can broadcast successfully to the application.

Why Not Use Packages?

You shouldn't use runtime packages unless you are sure that other applications will be using these packages. Otherwise, these packages will end up using more disk space than if you were to just compile the source code into your final executable. Why is this so? If you create a packaged application resulting in a code reduction from 200KB to roughly 30KB, it might seem like you've saved quite a bit of space. However, you still have to distribute your packages and possibly even the `Vcl60.dcp` package, which is roughly 2MB in size. You can see that this isn't quite the saving you had hoped for. Our point is that you should use packages to share code when that code will be used by multiple executables. Note that this only applies to runtime packages. If you are a component writer, you must provide a design package that contains the component you want to make available to the Delphi IDE.

Types of Packages

Four types of packages are available for you to create and use:

- Runtime package—Runtime packages contain code, components, and so on needed by an application at runtime. If you write an application that depends on a particular runtime package, the application won't run in the absence of that package.
- Design package—Design packages contain components, property/component editors, experts, and so on necessary for application design in the Delphi IDE. This type of package is used only by Delphi and is never distributed with your applications.
- Runtime and design package—A package that is both design- and runtime-enabled is typically used when there are no design-specific elements such as property/component editors and experts. You can create this type of package to simplify application development and deployment. However, if this package does contain design elements, its runtime use will carry the extra baggage of the design support in your deployed applications. In the event of many design time elements, we recommend creating both a design and runtime package to separate design-specific elements when they are present.
- Neither runtime nor design package—This rare breed of package is intended to be used only by other packages and isn't intended to be referenced directly by an application or used in the design environment. This implies that packages can use or include other packages.

Package Files

Table 14.1 lists and describes the types of package-specific files based on their file extensions.

TABLE 14.1 Package Files

File Extension	*File Type*	*Description*
`.dpk`	Package source file	This file is created when you invoke the Package Editor. You can think of this as you might think of the `.dpr` file for a Delphi project.
`.dcp`	Runtime/Design package symbol file	This is the compiled version of the package that contains the symbol information for the package and its units. Additionally, there is header information required by the Delphi IDE.
`.dcu`	Compiled unit	A compiled version of a unit contained in a package. One `.dcu` file will be created for each unit contained in the package.

TABLE 14.1 Continued

File Extension	*File Type*	*Description*
`.bpl`	Runtime/Design	This is the runtime or design package library package, equivalent to a Windows DLL. If this is a runtime package, you will distribute the file along with your applications (if they are enabled for runtime packages). If this file represents a design package, you will distribute it along with its runtime partner to programmers that will use it to write programs. Note that if you aren't distributing source code, you must distribute the corresponding `.dcp` files.

Using Runtime Packages

To use runtime packages in your Delphi applications, simply check the Build With Runtime Packages check box found in the Project, Options dialog on the Packages page. The next time you build your application after this option is selected, your application will be linked dynamically to runtime packages instead of having units linked statically into your `.exe` or `.dll`. The result will be a much more svelte application (although bear in mind that you will have to deploy the necessary packages with your application).

Installing Packages into the Delphi IDE

It's sometimes necessary to install a package into the Delphi IDE. This would be the case if you were to acquire a third-party component set or Delphi add-in that didn't do this during the install.

This being the case, you must first place the package files in their appropriate location. Table 14.2 shows where package files are typically located.

TABLE 14.2 Package File Locations

Package File	*Location*
Runtime packages (`*.bpl`)	Runtime package files should be placed in the `\Windows\System\` directory (Windows 95/98) or `\WinNT\System32\` directory (Windows NT/2000).
Design packages (`*.bpl`)	Because it is possible that you will obtain several packages from various vendors, design packages should be placed in a common directory where they can be properly managed. For example, create a `\PKG` directory off your `\Delphi 6\` directory and place design packages in that location.

TABLE 14.2 Continued

Package File	*Location*
Package symbol files (`*.dcp`)	You can place package symbol files in the same location as design package files (`*.bpl`).
Compiled units (`*.dcu`)	You must distribute compiled units if you are distributing design packages without source. We recommend keeping DCUs from third-party vendors in a directory similar to the `\Delphi 6\Lib` directory. For example, you can create the directory `\Delphi 6\3PrtyLib` in which third-party components' `*.dcu`s will reside. Your search path will have to include this directory.

To install a package, you simply invoke the Packages page of the Project Options dialog box by selecting Component, Install Packages from the Delphi 6 menu.

By clicking the Add button, you can select the specific `.bpl` file. Upon doing so, this file will become the selected file on the Project page. When you click OK, the new package is installed into the Delphi IDE. If this package contains components, you will see the new component page on the Component Palette along with any newly installed components.

Creating Packages

Before creating a package, you'll need to decide on a few things. First, you need to know what type of package you're going to create (runtime, design, and so on). This will be based on one or more of the scenarios that we present momentarily. Second, you need to know what you intend on naming your newly created package and where you want to store the package project. Keep in mind that the directory where your deployed package exists will probably not be the same as where you create your package. Finally, you need to know which units your package will contain and which other packages your new package will require.

The Package Editor

Packages are most commonly created using the Package Editor, which you invoke by selecting the Packages icon from the Object Repository. (Select File, New, Other from the Delphi main menu.) You'll notice that the Package Editor contains two folders: Contains and Requires.

The Contains Folder

In the Contains folder, you specify units that need to be compiled into your new package. There are a few rules for placing units into the Contains page of a package:

- The unit must not be listed in the `contains` clause of another package or `uses` clause of a unit within another package, which will be loaded concurrently with the package the unit is to be contained in.
- The units listed in the `contains` clause of a package, either directly or indirectly (they exist in `uses` clauses of units listed in the package's `contains` clause), cannot be listed in the package's `requires` clause. This is because these units are already bound to the package when it is compiled.
- You cannot list a unit in a package's `contains` clause if it is already listed in the `contains` clause of another package used by the sameapplication.

The Requires Folder

In the Requires folder, you specify other packages that are required by the new package. This is similar to the `uses` clause of a Delphi unit. In most cases, any packages you create will have `VCL60`—the package that hosts Delphi's standard VCL components—in its `requires` clause. The typical arrangement here, for example, is that you place all your components into a runtime package. Then you create a design package that includes the runtime package in its `requires` clause. There are a few rules for placing packages on the Requires folder of another package:

- Avoid circular references—`Package1` cannot have `Package1` in its `requires` clause, nor can it contain another package that has `Package1` in its `requires` clause.
- The chain of references must not refer back to a package previously referenced in the chain.

The Package Editor has a toolbar and context-sensitive menus. Refer to the Delphi 6 online help under "Package Editor" for an explanation of what these buttons do. We won't repeat that information here.

Package Design Scenarios

Earlier we said that you must know what type of package you want to create based on a particular scenario. In this section, we're going to present four possible scenarios in which you would use design and/or runtime packages.

Scenario 1—Design and Runtime Packages for Components

The Design and Runtime Packages for Components scenario is the case in which you are a component writer and one or both of the following conditions apply:

- You want Delphi programmers to be able to compile/link your components right into their applications or to distribute them separately along with their applications.
- You have a component package, and you don't want to force your users to have to compile design features (component/property editors and so on) into their application code.

Given this scenario, you would create both a design and runtime package. Figure 14.1 depicts this arrangement. As the figure illustrates, the design package (`ddgDT60.dpk`) encompasses both the design features (property and component editors) and the runtime package (`ddgRT60.dpk`). The runtime package (ddgRT60`.dpk`) includes only your components. This arrangement is accomplished by listing the runtime package into the `requires` section of the design package, as shown in Figure 14.1.

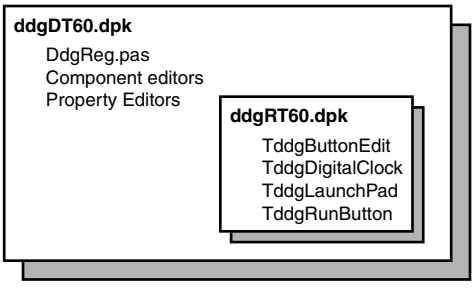

FIGURE 14.1
Design packages hosts design elements and runtime packages.

You must also apply the appropriate usage options for each package before compiling that package. You do this from the Package Options dialog box. (You access the Package Options dialog box by right-clicking within the Package Editor to invoke the local menu. Select Options to get to the dialog box.) For the runtime package, `DdgRT60.dpk`, the usage option should be set to Runtime Only. This ensures that the package cannot be installed into the IDE as a design package (see the sidebar "Component Security" later in this chapter). For the design package, `DdgDT60.dpk`, the usage option Design Time Only should be selected. This enables users to install the package into the Delphi IDE, yet prevents them from using the package as a runtime package.

Adding the runtime package to the design package doesn't make the components contained in the runtime package available to the Delphi IDE yet. You must still register your components with the IDE. As you already know, whenever you create a component, Delphi automatically inserts a `Register()` procedure into the component unit, which in turn calls the `RegisterComponents()` procedure. `RegisterComponents()` is the procedure that actually registers your component with the Delphi IDE when you install the component. When working with packages, the recommended approach is to move the `Register()` procedure from the component unit into a separate registration unit. This registration unit registers all your components by calling `RegisterComponents()`. This not only makes it easier for you to manage the registration of your components, but it also prevents anyone from being able to install and use your runtime package illegally because the components won't be available to the Delphi IDE.

As an example, the components used in this book are hosted by the runtime package `DdgRT60.dpk`. The property editors, component editors, and registration unit (`DdgReg.pas`) for our components exist in the design package `DdgDT60.dpk`. `DdgDT60.dpk` also includes `DdgRT60.dpk` in its `requires` clause. Listing 14.1 shows what our registration unit looks like.

LISTING 14.1 Registration Unit for Delphi 6 Developer's Guide Components

```
unit DDGReg;

interface

procedure Register;

implementation

uses Classes, ExptIntf, DsgnIntf, TrayIcon, AppBars, ABExpt, Worthless,
  RunBtn, PwDlg, Planets, LbTab, HalfMin, DDGClock, ExMemo, MemView,
  Marquee, PlanetPE, RunBtnPE, CompEdit, DefProp, Wavez,
  WavezEd, LnchPad, LPadPE, Cards, ButtonEdit, Planet, DrwPnel;

procedure Register;
begin

  // Register the components.
  RegisterComponents('DDG',
  [ TddgTrayNotifyIcon, TddgDigitalClock, TddgHalfMinute, tddgButtonEdit,
    TddgExtendedMemo, TddgTabListbox, TddgRunButton, TddgLaunchPad,
    TddgMemView, TddgMarquee, TddgWaveFile, TddgCard, TddgPasswordDialog,
    TddgPlanet, TddgPlanets, TddgWorthLess, TddgDrawPanel,
    TComponentEditorSample, TDefinePropTest]);


  // Register any property editors.
  RegisterPropertyEditor(TypeInfo(TRunButtons), TddgLaunchPad, '',
    TRunButtonsProperty);
  RegisterPropertyEditor(TypeInfo(TWaveFileString), TddgWaveFile, 'WaveName',
    TWaveFileStringProperty);
  RegisterComponentEditor(TddgWaveFile, TWaveEditor);
  RegisterComponentEditor(TComponentEditorSample, TSampleEditor);
  RegisterPropertyEditor(TypeInfo(TPlanetName), TddgPlanet,
   'PlanetName', TPlanetNameProperty);
  RegisterPropertyEditor(TypeInfo(TCommandLine), TddgRunButton, '',
    TCommandLineProperty);
```

LISTING 14.1 Continued

```
  // Register any custom modules, library experts.
  RegisterCustomModule(TAppBar, TCustomModule);
  RegisterLibraryExpert(TAppBarExpert.Create);

end;

end.
```

Component Security

It is possible for someone to register your components, even though he has only your runtime package. He would do this by creating his own registration unit in which he would register your components. He would then add this unit to a separate package that would also have your runtime package in the `requires` clause. After he installs this new package into the Delphi IDE, your components will appear on the Component Palette. However, it is still not possible to compile any applications using your components because the required `*.dcu` files for your component units will be missing.

Package Distribution

When distributing your packages to component writers without the source code, you must distribute both compiled packages, `DdgDT6.bpl` and `DdgRT6.bpl`, both `*.dcp` files, and any compiled units (`*.dcu`) necessary to compile your components. Programmers using your components who want their applications' runtime packages enabled must distribute the `DdgRT6.bpl` package along with their applications and any other runtime package that they might be using.

Scenario 2—Design Package Only for Components

The Design Package Only for Components scenario is the case in which you want to distribute components that you don't want to be distributed in runtime packages. In this case, you will include the components, component editors, property editors, component registration unit, and so on in one package file.

Package Distribution

When distributing your package to component writers without the source code, you must distribute the compiled package, `DdgDT6.bpl`, the `DdgDT6.dcp` file, and any compiled units (`*.dcu`) necessary to compile your components. Programmers using your components must compile your components into their applications. They will not be distributing any of your components as runtime packages.

Scenario 3—Design Features Only (No Components) IDE Enhancements

The Design Features Only (No Components) IDE Enhancements scenario is the case in which you are providing enhancements to the Delphi IDE, such as experts. For this scenario, you will register your expert with the IDE in your registration unit. The distribution for this scenario is simple; you only have to distribute the compiled `*.bpl` file.

Scenario 4—Application Partitioning

The Application Partitioning scenario is the case in which you want to partition your application into logical pieces, each of which can be distributed separately. You might want to do this for several reasons:

- This scenario is easier to maintain.
- Users can purchase only the needed functionality when they need it. Later, when they need added functionality, they can download the necessary package only, which will be much smaller than downloading the entire application.
- You can provide fixes (patches) to parts of the application more easily without requiring users to obtain a new version of the application altogether.

In this scenario, you will provide only the `*.bpl` files required by your application. This scenario is similar to the last with the difference being that instead of providing a package for the Delphi IDE, you will be providing a package for your own application. When partitioning your applications as such, you must pay attention to the issues regarding package versioning that we discuss in the next section.

Package Versioning

Package versioning is a topic that isn't well understood. You can think of package versioning in much the same way as you think of unit versioning. That is, any package you provide for your application must be compiled using the same Delphi version used to compile the application. Therefore, you cannot provide a package written in Delphi 6 to be used by an application written in Delphi 5. The Borland developers refer to the version of a package as a *code base*. So a package written in Delphi 6 has a code base of 6.0. This concept should influence the naming convention that you use for your package files.

Package Compiler Directives

There are some specific compiler directives that you can insert into the source code of your packages. Some of these directives are specific to units that are being packaged; others are specific to the package file. These directives are listed and described in Tables 14.3 and 14.4.

TABLE 14.3 Compiler Directives for Units Being Packaged

Directive	*Meaning*
`{$G}` or `{IMPORTEDDATA OFF}`	Use this when you want to prevent the unit from being packaged—when you want it to be linked directly to the application. Contrast this to the `{$WEAKPACKAGEUNIT}` directive, which allows a unit to be included in a package but whose code gets statically linked to the application.
`{$DENYPACKAGEUNIT}`	Same as `{$G}`.
`{$WEAKPACKAGEUNIT}`	See the section "More on `{$WEAKPACKAGEUNIT}`."

TABLE 14.4 Compiler Directives for the Package `.dpk` File

Directive	*Meaning*
`{$DESIGNONLY ON}`	Compiles the package as a design-time only package.
`{$RUNONLY ON}`	Compiles the package as a runtime only package.
`{$IMPLICITBUILD OFF}`	Prevents the package from being rebuilt later. Use this option when the package isn't changed frequently.

More on `{$WEAKPACKAGEUNIT}`

The concept of a weak package is simple. Basically, it is used where your package might be referencing libraries (DLLs) that might not be present. For example, the package `Vcl60` makes calls to the core Win32 API included with the Windows operating system. Many of these calls exist in DLLs that aren't present on every machine. These calls are exposed by units that contain the `{$WEAKPACKAGEUNIT}` directive. By including this directive, you keep the unit's source code in the package but place it into the DCP file rather than in the BPL file (think of a DCP as a DCU and a BPL as a DLL). Therefore, any references to functions of these weakly packaged units get statically linked to the application rather than dynamically referenced through the package.

The `{$WEAKPACKAGEUNIT}` directive is one that you will rarely use, if at all. It was created out of necessity by the Delphi developers to handle a specific situation. The problem exists if there are two components, each in a separate package and referencing the same interface unit of a DLL. When an application uses both of the components, this causes two instances of the DLL to be loaded, which raises havoc with initialization and global variable referencing. The solution was to provide the interface unit into one of the standard Delphi packages such as `Vcl60.bpl`. However, this raises the other problem for specialized DLLs that may not be present such as `PENWIN.DLL`. If `Vcl60.bpl` contains the interface unit for a DLL that isn't present, it will render

Vcl60.bpl, and Delphi for that matter, unusable. The Delphi developers addressed this by allowing Vcl60.bpl to contain the interface unit in a single package, but to make it statically linked when used and not dynamically loaded whenever Vcl60 is used with the Delphi IDE.

You'll most likely never have to use this directive, unless you anticipate a similar scenario that the Delphi developers faced or if you want to make certain that a particular unit is included with a package but statically linked to the using application. A reason for the latter might be for optimization purposes. Note that any units that are weakly packaged cannot have global variables or code in their initialization/finalization sections. You must also distribute any *.dcu files for weakly packaged units along with your packages.

Package Naming Conventions

Earlier we said that the package versioning issue should influence how you name your packages. There isn't a set rule for how to name your packages, but we suggest using a naming convention that incorporates the code base into the package's name. For example, the components for this book are contained in a runtime package whose name contains the 6 qualifier for Delphi 6 (DdgRT6.dpk). The same goes for the design package (DdgDT6.dpk). A previous version of the package would be DdgRT5.dpk. By using such a convention, you will prevent any confusion for your package users as to which version of the package they have and as to which version of the Delphi compiler applies to them. Note that our package name starts with a 3-character author/company identifier, followed by RT to indicate a runtime package and DT to signify a design time package. You can follow whatever naming convention you like. Just be consistent and use the recommended inclusion of the Delphi version into your package name.

Extensible Applications Using Runtime (Add-In) Packages

Add-in packages allow you to partition your applications into modules and to distribute those modules separately from the main application. This is useful because it allows you to extend the functionality of your application without having to recompile/redesign the entire application. This, however, requires careful architectural design planning. Although it is beyond the scope of this book to go into such design issues, our discussion will illustrate how to take advantage of this powerful capability.

Generating Add-In Forms

The application is partitioned into three logical pieces: the main application (ChildTest.exe), the TChildForm package (AIChildFrm6.bpl), and the concrete TChildForm descendant classes, each residing in its own package.

The package AIChildFrm6.bpl contains the base TChildForm class. The other packages contain descendant TChildForm classes or *concrete* TChildForms. We will refer to these packages as the base package and concrete packages, respectively.

The main application uses the abstract package (AIChildFrm6.bpl). Each concrete package also uses the abstract package. In order for this to work properly the main application must be compiled with runtime packages including the AIChildFrm6.dcp package. Likewise, each concrete package must require the AIChildFrm6.dcp package. We will not list the TChildForm source nor the concrete descendants to each TChildForm descendant unit, which must include initialization and finalization blocks that look like this:

```
initialization
  RegisterClass(TCF2Form);
finalization
  UnRegisterClass(TCF2Form);
```

The call to RegisterClass() is necessary to make the TChildForm descendant class available to the main application's streaming system when the main application loads its package. This is similar to how RegisterComponents() makes available components to the Delphi IDE. When the package is unloaded, the call to UnRegisterClass() is required to remove the registered class. Note, however, that RegisterClass() only makes the class available to the main application. The main application still doesn't know of the classname. So how does the main application create an instance of a class whose classname is unknown? Isn't the intent of this exercise to make these forms available to the main application without having to hard-code their classnames into the main applications source? Listing 14.2 shows the source code to the main application's main form where we will highlight how we accomplish add-in forms with add-in packages.

LISTING 14.2 Main Form to the Main Application Using Add-In Packages

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, ChildFrm, Menus;

const
  { Child form registration location in the Windows Registry. }
  cAddInIniFile  = 'AddIn.ini';
  cCFRegSection  = 'ChildForms';  // Module initialization data section

  FMainCaption   = 'Delphi 6 Developer''s Guide Child Form Demo';
```

LISTING 14.2 Continued

```
type

  TChildFormClass = class of TChildForm;

  TMainForm = class(TForm)
    pnlMain: TPanel;
    Splitter1: TSplitter;
    pnlParent: TPanel;
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiHelp: TMenuItem;
    mmiForms: TMenuItem;
    procedure mmiExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    // reference to the child form.
    FChildForm: TChildForm;
    // a list of available child forms used to build a menu.
    FChildFormList: TStringList;
    // Index to the Close Form menu which shifts position.
    FCloseFormIndex: Integer;
    // Handle to the currently loaded package.
    FCurrentModuleHandle: HModule;
    // method to create menus for available child forms.
    procedure CreateChildFormMenus;
    // Handler to load a child form and its package.
    procedure LoadChildFormOnClick(Sender: TObject);
    // Handler to unload a child form and its package.
    procedure CloseFormOnClick(Sender: TObject);
    // Method to retrieve the classname for a TChildForm descendant
    function GetChildFormClassName(const AModuleName: String): String;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses IniFiles;

{$R *.DFM}
```

LISTING 14.2 Continued

```
function RemoveExt(const AFileName: String): String;
{ Helper function to remove the extension from a file name. }
begin
  if Pos('.', AFileName) <> 0 then
    Result := Copy(AFileName, 1,  Pos('.', AFileName)-1)
  else
    Result := AFileName;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FChildFormList := TStringList.Create;
  CreateChildFormMenus;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FChildFormList.Free;
  // Unload any loaded child forms.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);
end;

procedure TMainForm.CreateChildFormMenus;
{ All available child forms are registered in the Windows Registry.
  Here, we use this information to create menu items for loading each of the
  child forms. }
var
  IniFile: TIniFile;
  MenuItem: TMenuItem;
  i:  integer;
begin
  inherited;

  { Retrieve a list of all child forms and build a menu based on the
    entries in the registry. }
  IniFile :=
TIniFile.Create(ExtractFilePath(Application.ExeName)+cAddInIniFile);
  try
```

LISTING 14.2 Continued

```
    IniFile.ReadSectionValues(cCFRegSection, FChildFormList);
  finally
    IniFile.Free;
  end;

  { Add Menu items for each module. Note the mmMain.AutoHotKeys property must
    bet set to maAutomatic }

  for i := 0 to FChildFormList.Count - 1 do
  begin
    MenuItem := TMenuItem.Create(mmMain);
    MenuItem.Caption := FChildFormList.Names[i];
    MenuItem.OnClick := LoadChildFormOnClick;
    mmiForms.Add(MenuItem);
  end;

  // Create Separator
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '-';
  mmiForms.Add(MenuItem);

  // Create Close Module menu item
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '&Close Form';
  MenuItem.OnClick := CloseFormOnClick;
  MenuItem.Enabled := False;
  mmiForms.Add(MenuItem);

  { Save a reference to the index of the menu item required to
    close a child form. This will be referred to in another method. }
  FCloseFormIndex := MenuItem.MenuIndex;
end;

procedure TMainForm.LoadChildFormOnClick(Sender: TObject);
var
  ChildFormClassName: String;
  ChildFormClass: TChildFormClass;
  ChildFormName: String;
  ChildFormPackage: String;
begin

  // The menu caption represents the module name.
  ChildFormName := (Sender as TMenuItem).Caption;
  // Get the actual Package file name.
  ChildFormPackage := FChildFormList.Values[ChildFormName];
```

LISTING 14.2 Continued

```
  // Unload any previously loaded packages.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);

  try
    // Load the specified package
    FCurrentModuleHandle := LoadPackage(ChildFormPackage);

    // Return the classname that needs to be created
    ChildFormClassName :=  GetChildFormClassName(ChildFormPackage);

    { Create an instance of the class using the FindClass() procedure. Note,
      this requires that the class already be registered with the streaming
      system using RegisterClass(). This is done in the child form
      initialization section for each child form package. }
    ChildFormClass := TChildFormClass(FindClass(ChildFormClassName));
    FChildForm := ChildFormClass.Create(self, pnlParent);
    Caption := FChildForm.GetCaption;

    { Merge child form menus with the main menu }
    if FChildForm.GetMainMenu <> nil then
      mmMain.Merge(FChildForm.GetMainMenu);

    FChildForm.Show;

    mmiForms[FCloseFormIndex].Enabled := True;
  except
    on E: Exception do
    begin
      CloseFormOnClick(nil);
      raise;
    end;
  end;
end;

function TMainForm.GetChildFormClassName(const AModuleName: String): String;
{ The Actual class name of the TChildForm implementation resides in the
  registry. This method retrieves that class name. }
var
  IniFile:  TIniFile;
begin
  IniFile :=
TIniFile.Create(ExtractFilePath(Application.ExeName)+cAddInIniFile);
  try
```

LISTING 14.2 Continued

```
    Result := IniFile.ReadString(RemoveExt(AModuleName), 'ClassName',
      EmptyStr);
  finally
    IniFile.Free;
  end;
end;

procedure TMainForm.CloseFormOnClick(Sender: TObject);
begin
  if FCurrentModuleHandle <> 0 then
  begin
    if FChildForm <> nil then
    begin
      FChildForm.Free;
      FChildForm := nil;
    end;

    // Unregister any classes provided by the module
    UnRegisterModuleClasses(FCurrentModuleHandle);
    // Unload the child form package
    UnloadPackage(FCurrentModuleHandle);

    FCurrentModuleHandle := 0;
    mmiForms[FCloseFormIndex].Enabled := False;
    Caption := FMainCaption;
  end;
end;

end.
```


The application's logic is actually very simple. It uses the system registry to determine which packages are available, the menu captions to use when building menus for loading each package, and the classname of the form contained in each package.

The `LoadChildFormOnClick()` event handler is where most of the work is performed. After determining the package filename, the method loads the package using the `LoadPackage()` function. The `LoadPackage()` function is basically the same thing as `LoadLibrary()` for DLLs. The method then determines the classname for the form contained in the loaded package.

In order to create a class, you require a class reference like `TButton` or `TForm1`. However, this main application doesn't have the hard-coded classname of the concrete `TChildForms`, so this is why we retrieve the classname from the system registry. The main application can pass this classname to the `FindClass()` function to return a class reference for the specified class that

has already been registered with the streaming system. Remember, we did this in the initialization section of the concrete form's unit that is called when the package is loaded. We then create the class with the lines:

```
ChildFormClass := TChildFormClass(FindClass(ChildFormClassName));
FChildForm := ChildFormClass.Create(self, pnlParent);
```

NOTE

A class reference is simply an area in memory that contains information about a class. This is the same as a type-definition for a class. It gets into memory when the class is registered with the VCL streaming system; when the `RegisterClass()` function is called. The `FindClass()` function locates the area of memory for a class of a specified name and returns a pointer to that location. This isn't the same as a class instance. Class instances are usually created when the constructor, a class function (see Chapter 2, "The Object Pascal Language"), is called.

The variable `ChildFormClass` is a pre-declared class reference to `TChildForm` and can polymorphically refer to a class reference for a `TChildForm` descendant.

The `CloseFormOnClick()` event handler simply closes the child form and unloads its package. The rest of the code is basically set up code to create the package menus and to read the information from the INI file.

Using this technique, you can create very extensible and loosely coupled application frameworks.

Exporting Functions from Packages

Given that packages are simply enhanced DLLs, it seems that you should be able to export functions and procedures from packages just as you can from DLLs. Well, you can. In this section, we'll show you how to use packages in the same way.

Launching a Form from a Package Function

Listing 14.3 is a unit contained inside of a package.

LISTING 14.3 Package Unit with Two Exported Functions

```
unit FunkFrm;

interface
```

LISTING 14.3 Continued

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type

  TFunkForm = class(TForm)
    Label1: TLabel;
    Button1: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

// Declare the package functions using the StdCall calling convention
procedure FunkForm; stdcall;
function AddEm(Op1, Op2: Integer): Integer; stdcall;

// Export the functions.
exports
  FunkForm,
  AddEm;

implementation

{$R *.dfm}

procedure FunkForm;
var
  FunkForm: TFunkForm;
begin
  FunkForm := TFunkForm.Create(Application);
  try
    FunkForm.ShowModal;
  finally
    FunkForm.Free;
  end;
end;

function AddEm(Op1, Op2: Integer): Integer;
begin
  Result := Op1+Op2;
end;

end.
```

The procedure `FunkForm()` simply displays the form declared in the unit as a modal form; nothing clever here. `AdEm()` is a function that takes two operands and returns their sum. Notice that the functions are declared in the interface section of this unit using the `StdCall` calling convention.

Listing 14.4 is an application that demonstrates how to invoke a function from a package.

LISTING 14.4 Demo Application

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Mask;

const
  cFunkForm = 'FunkForm';
  cAddEm    = 'AddEm';

type
  TForm1 = class(TForm)
    btnPkgForm: TButton;
    meOp1: TMaskEdit;
    meOp2: TMaskEdit;
    btnAdd: TButton;
    lblPlus: TLabel;
    lblEquals: TLabel;
    lblResult: TLabel;
    procedure btnAddClick(Sender: TObject);
    procedure btnPkgFormClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  // Defined the method signatures
  TAddEmProc = function(Op1, Op2: Integer): integer; stdcall;
  TFunkFormProc = procedure; stdcall;

var
  Form1: TForm1;

implementation
```

LISTING 14.4 Continued

```
{$R *.dfm}

procedure TForm1.btnAddClick(Sender: TObject);
var
  PackageModule: THandle;
  AddEmProc: TAddEmProc;
  Rslt: Integer;
  Op1, Op2: integer;
begin
  PackageModule := LoadPackage('ddgPackFunk.bpl');
  try

    @AddEmProc := GetProcAddress(PackageModule, PChar(cAddEm));
    if not (@AddEmProc = nil) then
    begin
      Op1 := StrToInt(meOp1.Text);
      Op2 := StrToInt(meOp2.Text);

      Rslt := AddEmProc(Op1, Op2);
      lblResult.Caption := IntToStr(Rslt);
    end;

  finally
    UnloadPackage(PackageModule);
  end;
end;

procedure TForm1.btnPkgFormClick(Sender: TObject);
var
  PackageModule: THandle;
  FunkFormProc: TFunkFormProc;
begin
  PackageModule := LoadPackage('ddgPackFunk.bpl');
  try
    @FunkFormProc := GetProcAddress(PackageModule, PChar(cFunkForm));
    if not (@FunkFormProc = nil) then
      FunkFormProc;
  finally
    UnloadPackage(PackageModule);
  end;
end;

end.
```

First notice that we had to declare the two procedural types, `TAddEmProc` and `TFunkFormProc`. These are declared exactly as they exist in the package.

We'll discuss the `btnPkgFormClick()` event handler first. This code should look familiar from Chapter 6, "Dynamic Link Libraries." Instead of making a `LoadLibrary()` call, we're using `LoadPackage()`. In fact, `LoadPackage()` ends up calling `LoadLibrary()`. Next, we retrieve the reference to the procedure using the `GetProcAddress()` function. You can refer back to Chapter 6 if you need to know more about this function. The `cFunkForm` constant is the same name as the function name in the package.

You can see that the method of exporting functions and procedures from packages is almost exactly the same as exporting from dynamic link libraries.

Obtaining Information About a Package

It is possible to query a package for information about which units it contains and which packages it requires. Two functions are used to do this: `EnumModules()` and `GetPackageInfo()`. Both of these functions require callback functions. Listing 14.5 illustrates the use of these functions. You'll find this demo on the CD.

LISTING 14.5 Package Information Demo

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, DBXpress, DB, SqlExpr, DBTables;

type
  TForm1 = class(TForm)
    Button1: TButton;
    TreeView1: TTreeView;
    Table1: TTable;
    SQLConnection1: TSQLConnection;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

LISTING 14.5 Continued

```
implementation

{$R *.dfm}

type
  TNodeHolder = class
    ContainsNode: TTreeNode;
    RequiresNode: TTreeNode;
  end;

procedure RealizeLength(var S: string);
begin
  SetLength(S, StrLen(PChar(S)));
end;

procedure PackageInfoProc(const Name: string; NameType:
  TNameType; Flags: Byte; Param: Pointer);
var
  NodeHolder: TNodeHolder;
  TempStr: String;
begin
  with Form1.TreeView1.Items do
  begin

    TempStr := EmptyStr;

    if (Flags and ufMainUnit) <> 0 then
      TempStr := 'Main unit'
    else if (Flags and ufPackageUnit) <> 0 then
      TempStr := 'Package unit' else
    if (Flags and ufWeakUnit) <> 0 then
      TempStr := 'Weak unit';

    if TempStr <> EmptyStr then
      TempStr := Format(' (%s)', [TempStr]);

    NodeHolder := TNodeHolder(Param);
    case NameType of
      ntContainsUnit: AddChild(NodeHolder.ContainsNode,
        Format('%s %s', [Name,TempStr]));
      ntRequiresPackage: AddChild(NodeHolder.RequiresNode, Name);
    end; // case
  end;
end;
```

LISTING 14.5 Continued

```
function EnumModuleProc(HInstance: integer; Data: Pointer): Boolean;
var
  ModFileName: String;
  ModNode: TTreeNode;
  ContainsNode: TTreeNode;
  RequiresNode: TTreeNode;
  ModDesc: String;
  Flags: Integer;
  NodeHolder: TNodeHolder;

begin
  with Form1.TreeView1 do
  begin
    SetLength(ModFileName, 255);
    GetModuleFileName(HInstance, PChar(ModFileName), 255);
    RealizeLength(ModFileName);
    ModNode := Items.Add(nil, ModFileName);

    ModDesc := GetPackageDescription(PChar(ModFileName));
    ContainsNode := Items.AddChild(ModNode, 'Contains');
    RequiresNode := Items.Addchild(ModNode, 'Requires');


    if ModDesc <> EmptyStr then
    begin

      NodeHolder := TNodeHolder.Create;
      try
        NodeHolder.ContainsNode := ContainsNode;
        NodeHolder.RequiresNode := RequiresNode;

        GetPackageInfo(HInstance, NodeHolder, Flags, PackageInfoProc);
      finally
        NodeHolder.Free;
      end;

      Items.AddChild(ModNode, ModDesc);

      if Flags and pfDesignOnly = pfDesignOnly then
        Items.AddChild(ModNode, 'Design-time package');
      if Flags and pfRunOnly = pfRunOnly then
        Items.AddChild (ModNode, 'Run-time package');

    end;
```

LISTING 14.5 Continued

```
  end;
  Result := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  EnumModules(EnumModuleProc, nil);
end;

end.
```

`EnumModules()` is first called. It enumerates the executable and any packages in the executable. The callback function passed to `EnumModules()` is `EnumModuleProc()`. This function populates a `TTreeview` component with information about each package in the application. Much of the code is setup code for the `TTreeView` component. The function `GetPackageDescription()` returns the description string contained in the packages resource. The call to `GetPackageInfo()` passes the callback function `PackageInfoProc()`.

In `PackageInfoProc()`, we are able to process the information in the package's information table. This function is called for every unit included in the package and for every package required by the package. Here, we again populate the `TTreeview` component with this information by examining the values of the `Flags` parameter and the `NameType` parameter. For additional information, both of these are explained in the online help under "TPackageInfoProc."

This code demonstration is a modification of a demo from Marco Cantu's excellent book *Mastering Delphi 5*, a must for every Delphi library.

Summary

Packages are a key part of the Delphi/VCL architecture. By learning how to use packages for more then just component containment, you can develop very elegantly designed and loosely bound architectures.