CHAPTER

4

Application Frameworks and Design Concepts

IN THIS CHAPTER

This chapter is about Delphi project management and architecting. It shows you how to use forms properly in your applications as well as how to manipulate their behavioral and visual characteristics. The techniques discussed in this chapter include application startup/initialization procedures, form reuse/inheritance, and user interface enhancement. The text discusses the framework classes that make up Delphi 5 applications: `TApplication`, `TForm`, `TFrame`, and `TScreen`. We'll then show you why understanding these concepts is essential to properly architecting Delphi applications.

Understanding the Delphi Environment and Project Architecture

There are at least two important factors in properly building and managing Delphi 5 projects. The first is knowing the ins and outs of the development environment in which you create your projects. The second is having a solid understanding of the inherent architecture of the applications created with Delphi 5. This chapter doesn't walk you through the Delphi 5 environment (the Delphi documentation shows you how to work within that environment); instead, this chapter points out features of the Delphi 5 IDE that help you manage your projects more effectively. This chapter will also explain the architecture inherent in all Delphi applications. This will not only allow you to maximize the environment's features but also to properly use a solid architecture instead of fighting it—a common mistake among those who don't understand Delphi project architectures.

Our first suggestion is to become well acquainted with the Delphi 5 development environment. This book assumes that you're already familiar with the Delphi 5 IDE. Second, this book assumes that you've thoroughly read the Delphi 5 documentation (hint). However, you should navigate through each of the Delphi 5 menus and bring up each of its dialog boxes. When you see an option, setting, or action you're unsure of, bring up the online help and read through it. The time you spend doing this can prove interesting as well as insightful (not to mention that you'll learn how to navigate through the online help efficiently).

TIP

The Delphi 5 help system is without a doubt the most valuable and speedy reference you have at your disposal. It would be advantageous to learn how to use it to explore the thousands of help screens available.

Delphi 5 contains help on everything from how to use the Delphi 5 environment to details on the Win32 API and complex Win32 structures. You can get immediate help on a topic by typing the topic in the editor and, with the cursor still on the word you typed, pressing Ctrl+F1. The help screen appears immediately. Help is also available from the Delphi 5 dialog boxes by selecting the Help button or by pressing F1 when a particular component has focus. You can also navigate through help by simply selecting Help from Delphi 5's Help menu.

Files That Make Up a Delphi 5 Project

A Delphi 5 project is composed of several related files. Some files are created at design time as you define forms. Others aren't created until you compile the project. To manage a Delphi 5 project effectively, you must know the purpose of each of these files. Both the Delphi 5 documentation and the online help give detailed descriptions of the Delphi 5 project files. It's a good idea to review the documentation to ensure that you're familiar with these files before going on with this chapter.

The Project File

The *project file* is created at design time and has the extension `.dpr`. This file is the main program source file. The project file is where the main form and any automatically created forms are instantiated. You'll seldom have to edit this file except when performing program initialization routines, displaying a splash screen, or performing various other routines that must happen immediately when the program starts. The following code shows a typical project file:

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Pascal programmers will recognize this file as a standard Pascal program file. Notice that this file lists the form unit `Unit1` in the `uses` clause. Project files list all form units that belong to the project in the same manner. The following line refers to the project's resource file:

```
{$R *.RES}
```

This line tells the compiler to link the resource file that has the same name as the project file and an `.RES` extension to this project. The project resource file contains the program icon and version information.

Finally, the `begin..end` block is where the application's main code is executed. In this simple example, the main form, `Form1`, is created. When `Application.Run()` executes, `Form1` is displayed as the main form. You can add code in this block, as shown later in this chapter.

Project Unit Files

Units are Pascal source files with a `.pas` extension. There are basically three types of units files: form/data module and frame units, component units, and general-purpose units.

- *Form/data module and frame units* are units automatically generated by Delphi 5. There's one unit for each form/data module or frame you create. For example, you can't have two forms defined in one unit and use them both in the Form Designer. For the purpose of explaining form files, we won't make a distinction between forms, data modules, and frames.
- *Component units* are unit files created by you or Delphi 5 whenever you create a new component.
- *General-purpose units* are units you can create for data types, variables, procedures, and classes you want to make accessible to your applications.

Details about units are provided later in this chapter.

Form Files

A *form file* contains a binary representation of a form. Whenever you create a new form, Delphi 5 creates both a form file (with the extension `.dfm`) and a Pascal unit (with the extension `.pas`) for your new form. If you look at a form's unit file, you'll see the following line:

```
{$R *.DFM}
```

This line tells the compiler to link the corresponding form file (the form file that has the same name as the unit file and a `.DFM` extension) to the project.

You typically don't edit the form file itself (although it's possible to do so). You can load the form file into the Delphi 5 editor so that you can view or edit the text representation of this file. Select File, Open and then select the option to open only form files (`.dfm`). You can also do this by simply right-clicking the Form Designer and selecting View as Text from the pop-up menu. When you open the file, you see the text representation of the form.

Viewing the textual representation of the form is handy because you can see the nondefault property settings for the form and any components that exist on the form. One way you can edit the form file is to change a component type. For example, suppose that the form file contains this definition for a `TButton` component:

```
object Button1: TButton
    Left = 8
    Top = 8
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
```

If you change the line `object Button1: TButton` to `object Button1: TLabel`, you change the component type to a `TLabel` component. When you view the form, you see a label on the form and not a button.

NOTE

Changing component types in the form file might result in a property read error. For example, changing a `TButton` component (which has a `TabOrder` property) to a `TLabel` component (which doesn't have a `TabOrder` property) results in this error. However, there's no need for concern because Delphi will correct the reference to the property the next time the form is saved.

CAUTION

You must be extremely careful when you edit the form file. It's possible to corrupt it, which will prevent Delphi 5 from opening the form later.

NOTE

New to Delphi 5 is the ability to save forms in text file format. This was made possible to allow editing with other common tools such as `Notepad.exe`. Simply right-click the form to invoke the context menu and select Text DFM.

Resource Files

Resource files contain binary data, also called *resources*, that are linked to the application's executable file. The RES file automatically created by Delphi 5 contains the project's application icon, the application's version information, and other information. You can add resources to your application by creating a separate resource file and linking it to your project. You can create this resource file with a resource editor such as the Image Editor provided with Delphi 5 or the Resource Workshop.

CAUTION

Don't edit the resource file that Delphi creates automatically at compile time. Doing so will cause any changes to be lost the next time you compile. If you want to add resources to your application, create a separate resource file with a different name from that of your project file. Then link the new file to your project by using the `$R` directive, as shown in the following line:

```
   {$R MYRESFIL.RES}
```

Project Options and Desktop Settings Files

The *project options file* (with the extension `.dof`) is where the options specified from the Project, Options menu are saved. This file is created when you first save your project; the file is saved again with each subsequent save.

The *desktop options file* (with the extension `.dsk`) stores the options specified from the Tools, Environment Options menu for the desktop. Desktop option settings differ from project option settings in that project options are specific to a given project; desktop settings apply to the Delphi 5 environment.

TIP

A corrupt DSK or DOF file can cause unexpected results, such as a GPF, during compilation. If this happens, delete both the DOF and DSK files. They're regenerated when you save your project and when you exit Delphi 5; the IDE and project will revert to the default settings.

Backup Files

Delphi 5 creates *backup files* for the DPR project file and for any PAS units on the second and any subsequent saves. The backup files contain the last copy of the file before the save was performed. The project backup file has the extension `.~dp`. Unit backup files have the extension `.~pa`.

A binary backup of the DFM form file is also created after you've saved it for the second or subsequent time. This form file backup has a `.~df` extension.

You harm nothing if you delete any of these files—as long as you realize that you're deleting your last backup. Also, if you find that you prefer not to create these files at all, you can prevent Delphi from creating them by deselecting Create Backup File in the Editor Properties dialog box's Display page.

Package Files

Packages are simply DLLs that contain code that can be shared among many applications. However, packages are specific to Delphi in that they allow you to share components, classes, data, and code between modules. This means that you can now reduce the footprint of your applications drastically by using components residing in packages instead of linking them directly into your applications. Later chapters talk much more about packages. Package source files use the extension `.dpk` (short for *Delphi package*). When compiled, a BPL file is created (A .BPL file is simply a dll). This BPL may be composed of several units or DCU (*Delphi compiled units*) files, which can be any of the unit types previously mentioned. The binary image of a DPK file containing all included units and the package header has the extension `.dcp` (Delphi compiled package). Don't be concerned if this seems confusing now; we'll explain packages in more detail later.

Project Management Tips

There are several ways to optimize the development process by using techniques that facilitate better organization and code reuse. The next few sections offer some tips on these techniques.

One Project, One Directory

It's a good idea to manage your projects so that one project's files are separate from other projects' files. Doing so prevents one project from overwriting another project's files.

Notice that each project on the CD-ROM that accompanies this book is in its own directory. You should follow this approach and maintain each of your projects in its own directory.

File Naming Conventions

It's a good idea to establish a standard file naming convention for the files that make up your projects. You might take a look at the DDG Coding Standards Document included on the CD-ROM and used by the authors for the projects contained in this book. (See Chapter 6, "Delphi 5 Developer's Guide Coding Standards Document.")

Units for Sharing Code

You can share commonly used routines with other applications by placing such routines in units that can be accessed by multiple projects. Typically, you create a utility directory somewhere on your hard drive and place your units in that directory. When you need to access a particular function that exists in one of the units in that directory, you just place the unit's name in the `uses` clause of the unit/project file requiring access.

You must also add the utility directory's path to the search path on the Directories/Conditionals page in the Project Options dialog box. Doing so ensures that Delphi 5 knows where to find your utility units.

TIP

By using the Project Manager, you can add a unit from another directory to an existing project, which automatically takes care of adding the search path.

To explain how to use utility units, Listing 4.1 shows a small unit, `StrUtils.pas`, that contains a single string-utility function. In reality, such units would probably contain many more routines, but this suffices as an example. The comments explain the function's purpose.

LISTING 4.1 The `StrUtils.pas` Unit

```
unit strutils;
interface
function ShortStringAsPChar(var S: ShortString): PChar;
implementation
function ShortStringAsPChar(var S: ShortString): PChar;
{ This function null-terminates a short string so that it can be passed to
  functions that require PChar types. If string is longer than 254 chars, then
  it will be truncated to 254.
}
begin
  if Length(S) = High(S) then Dec(S[0]); { Truncate S if it's too long }
  S[Ord(Length(S)) + 1] := #0;           { Place null at end of string }
  Result := @S[1];                       { Return "PChar'd" string }
end;
end.
```

Suppose that you have a unit, `SomeUnit.Pas`, that requires the use of this function. Simply add `StrUtils` to the uses clause of the unit in need, as shown here:

```
unit SomeUnit;
interface
...
implementation
uses
  strutils;
...
end.
```

Also, you must ensure that Delphi 5 can find the unit `StrUtils.pas` by adding it to the search path from the Project, Options menu.

When you do this, you can use the function `ShortStringAsPChar()` anywhere in the implementation section of `SomeUnit.pas`. You must place `StrUtils` in the `uses` clause of all units that need access to the `ShortStringAsPChar()` function. It isn't enough to add `StrUtils` to only one unit in a project, or even to the project file (DPR) of the application to make the routine available throughout the entire application.

TIP

Because `ShortStringAsPChar()` is a handy function, it pays to place it in a utility unit where it can be reused by any application so that you don't have to remember how or where you last used it.

Units for Global Identifiers

Units are also useful for declaring global identifiers for your project. As mentioned earlier, a project typically consists of many units—form units, component units, and general-purpose units. What if you need a particular variable to be present and accessible to all units throughout the running of your application? The following steps show a simple way to create a unit to store these global identifiers:

1. Create a new unit in Delphi 5.
2. Give the unit a name indicating that it holds global identifiers for the application (for example, `Globals.Pas` or `ProjGlob.pas`).
3. Place the variables, types, and so on in the `interface` section of your global unit. These are the identifiers that will be accessible to other units in the application.
4. To make these identifiers accessible to a unit, just add the unit name to the `uses` clause of the unit that needs access (as described earlier in this chapter in the discussion about sharing code in units).

Making Forms Know About Other Forms

Just because each form is contained within its own unit doesn't mean that it can't access another form's variables, properties, and methods. Delphi generates code in the form's corresponding PAS file, declaring the instance of that form as a global variable. All that's required is that you add the name of the unit defining a particular form to the `uses` clause of the unit defining the form needing access. For example, if `Form1`, defined in `UNIT1.PAS`, needs access to `Form2`, defined in `UNIT2.PAS`, just add `UNIT2` to `UNIT1`'s `uses` clause:

```
unit Unit1;
interface
...
implementation
uses
  Unit2;
...
end.
```

Now `UNIT1` can refer to `Form2` anywhere in its `implementation` section.

NOTE

Form linking will ask you if you want to include `Unit2` in `Unit1`'s `uses` clause when you compile the project if you refer to the `Unit2`'s form (call it `Form2`); all that's necessary is to refer to `Form2` somewhere in `Unit1`.

Multiple Projects Management (Project Groups)

Often, a product is made up of multiple projects (projects that are dependent on one another). Examples of such projects are the separate tiers in a multitiered application. Also, DLLs to be used in other projects might be considered part of the overall project, even though DLLs are separate projects themselves.

Delphi 5 allows you to manage such project groups. The Project Manager allows you to combine several Delphi projects into one grouping called a *project group*. We won't go into to the details of using the Project Manager because Delphi's documentation already does this. We do want to emphasize how important it is to organize project groups and how the Project Manager helps you do this.

It's still important that each project live in its own directory and that all files specific to that project alone reside in the same directory. Any shared units, forms, and so on should be placed in a common directory that's accessed by the separate projects. For example, your directory structure might look something like this:

```
\DDGBugProduct
\DDGBugProduct\BugReportProject
\DDGBugProduct\BugAdminTool
\DDGBugProduct\CommonFiles
```

Given this structure, you have two separate directories for each Delphi project: `BugReportProject` and `BugAdminTool`. However, both of these projects may use forms and units that are common. You would place these files into the `CommonFiles` directory.

Organization is crucial in your development efforts, especially in a team development environment. It's highly recommended that you establish a standard before your team dives into creating a bunch of files that are going to be difficult to manage. You can use the Delphi Project Manager to help you understand your project-management structure.

The Framework Classes of a Delphi 5 Project

Most Delphi 5 applications have at least one instance of a `TForm` class. Also, Delphi 5 VCL applications will have only one instance of a `TApplication` class and a `TScreen` class. These three classes play important roles in managing the behavior of a Delphi 5 project. The following sections familiarize you with the roles of these classes so that you have the knowledge to modify their default behaviors when necessary.

The TForm Class

The `TForm` class is the focal point for Delphi 5 applications. In most cases, the entire application revolves around the main form. From the main form, you can launch other forms, usually as a result of a menu or button-click event. You might want Delphi 5 to create your forms automatically, in which case you don't have to worry about creating and destroying them. You may also choose to create the forms dynamically at runtime.

NOTE

Delphi can create applications that don't use forms (for example, console apps, services, and COM servers). Therefore, the `TForm` class is not always the focal point of your applications.

You can display the form to the end user by using one of two methods: modal or modeless. The method you choose depends on how you intend the user to interact with the form and other forms concurrently.

Displaying a Modal Form

A *modal form* is displayed so that the user can't access the rest of the application until he or she has dismissed the form. Modal forms are typically associated with dialog boxes, much like the dialog boxes in Delphi 5 itself. In fact, you'll probably use modal forms in most cases. To display a form as modal, simply call its `ShowModal()` method. The following code shows how you create an instance of a user-defined form, `TModalForm`, and then display it as a modal form:

```
Begin
  // Create ModalForm instance
  ModalForm := TModalForm.Create(Application);
  try
    if ModalForm.ShowModal = mrOk then     // Show form in modal state
     { do something };                     // Execute some code
  finally
    ModalForm.Free;                        // Free form instance
    ModalForm := nil;                      // Set form variable to nil
  end;
end;
```

This code shows how you would dynamically create an instance of `TModalForm` and assign it to the variable `ModalForm`. It's important to note that, if you create a form dynamically, you must remove it from the list of available forms from the Auto-Create list box in the Project Options dialog box. This dialog box is invoked by selecting Project, Options from the menu. If the form instance is already created, however, you can show it as a modal form just by calling the `ShowModal()` method. The surrounding code can be removed:

```
begin
  if ModalForm.ShowModal = mrOk then    // ModalForm is already created
    { do something }
end;
```

The `ShowModal()` method returns the value assigned to `ModalForm`'s `ModalResult` property. By default, `ModalResult` is zero, which is the value of the predefined constant `mrNone`. When you assign any nonzero value to `ModalResult`, the form is closed, and the assignment made to `ModalResult` is passed back to the calling routine through the `ShowModal()` method.

Buttons have a `ModalResult` property. You can assign a value to this property that's passed to the form's `ModalResult` property when the button is pressed. If this value is anything other than `mrNone`, the form will close, and the value passed back from the `ShowModal()` method will reflect that assigned to `ModalResult`.

You can also assign a value to the form's `ModalResult` property at runtime:

```
begin
  ModalForm.ModalResult := 100; // Assigning a value to ModalResult
  // causing form to close.
end;
```

Table 4.1 shows the predefined `ModalResult` values.

TABLE 4.1 ModalResult Values?

Constant	*Value*
`mrNone`	`0`
`mrOk`	`idOk`
`mrCancel`	`idCancel`

Constant	Value
mrAbort	idAbort
mrRetry	idRetry
mrIgnore	idIgnore
mrYes	idYes
mrNo	idNo
mrAll	mrNo+1

Launching Modeless Forms

You can launch a modeless form by calling its `Show()` method. Calling a modeless form differs from the modal method in that the user can switch between the modeless form and other forms in the application. The intent of modeless forms is to allow users to work with different parts of the application at the same time as the form is displayed. The following code shows how you can dynamically create a modeless form:

```
Begin
// Check for an instance of modeless first
  if not Assigned(Modeless) then
    Modeless := TModeless.Create(Application);  // Create form
  Modeless.Show                                 // Show form as non-modal
end;                                            // instance already exists
```

This code also shows how you prevent multiple instances of one form class from being created. Remember that a modeless form allows the user to interact with the rest of the application. Therefore, nothing prevents the user from selecting the menu option again to create another form instance of `TModeless`. It's important that you manage the creation and destruction of forms.

Here's an important note about form instances: When you close a modeless form—either by accessing the system menu or clicking the close button in the upper-right corner of the form—the form isn't actually freed from memory. The instance of the form still exists in memory until you close the main form (that is, the application). In the preceding code example, the `then` clause is executed only once, provided that the form is not autocreated. From that point on, the `else` clause is executed because the form instance always exists from its previous creation. This is fine if that's the way you want your application to function. However, if you want the form to be freed whenever the user closes it, you must provide code for the `OnClose` event handler for the form and set its `Action` parameter to `caFree`. This tells the VCL to free the form when it's closed:

```
procedure TModeless.FormClose(Sender: TObject;
  var Action: TCloseAction);
```

```
begin
  Action := caFree;  // Free the form instance when closed
end;
```

The preceding version of the code solves the issue of the form not being freed. There's another issue, however. You might have noticed that this line was used in the first snippet of code showing modeless forms:

```
if not Assigned(Modeless) then begin
```

This line checks for an instance of `TModeless` referenced by the `Modeless` variable. Actually, this really checks to see that `Modeless` is not `nil`. Although `Modeless` will be `nil` the first time you enter the routine, it won't be `nil` when you enter the routine a second time after having destroyed the form. The reason is because the VCL doesn't set the variable `Modeless` to `nil` when it's destroyed. Therefore, this is something you must do yourself.

Unlike with a modal form, you can't determine in code when the modeless form will be destroyed. Therefore, you can't destroy the form inside the routine that creates it. The user can close the form at any moment while running the application. Therefore, setting `Modeless` to `nil` must be a process of the `TModeless` class itself. The best place to do this is in the `OnDestroy` event handler for `TModeless`:

```
procedure TModeless.FormDestroy(Sender: TObject);
begin
  Modeless := nil; // Set the Modeless variable to nil when destroyed
end;
```

This ensures that the `Modeless` variable is set to `nil` every time it's destroyed, thus preventing the `Assigned()` method from failing. Keep in mind that it's up to you to ensure that only one instance of `TModeless` is created at a time, as shown in this routine.

CAUTION

Avoid the following pitfall when working with modeless forms:

```
begin
  Form1 := TForm1.Create(Application);
  Form1.Show;
end;
```

This code results in memory unnecessarily being consumed because, each time you create a form instance, you overwrite the previous instance referenced by `Form1`. Although you could refer to each instance of the form created through the `Screen.Forms` list, the practice shown in the preceding code is not recommended. Passing `nil` to the `Create()` constructor will result in no way to refer to the form instance pointer after the `Form1` instance variable is overwritten.

The project ModState.dpr on the accompanying CD-ROM illustrates using both modal and modeless forms.

Managing a Form's Icons and Borders

TForm has a BorderIcons property that's a set that may contain the following values: biSystemMenu, biMinimize, biMaximize, and biHelp. By setting any or all of these values to False, you can remove the system menu, the maximize button, the minimize button and the help button from the form. All forms have the Windows 95/98 close button.

You also can change the nonclient area of the form by changing the BorderStyle property. The BorderStyle property is defined as follows:

```
TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog,
➥bsSizeToolWin, bsToolWindow);
```

The BorderStyle property gives forms the following characteristics:

- bsDialog. Nonsizable border; close button only.
- bsNone. No border, nonsizable, and no buttons.
- bsSingle. Nonsizable border; all buttons available. If only one of the biMinimize and biMaximize buttons is set to False, both buttons appear on the form. However, the button set to False is disabled. If both are False, neither button appears on the form. If biSystemMenu is False, no buttons appear on the form.
- bsSizable. Sizable border. All buttons are available. The same circumstances exist for this option regarding buttons as with the bsSingle setting.
- bsSizeToolWin. Sizable border. Close button only and small caption bar.
- bsToolWindow. Nonsizable border. Close button only and small caption bar.

NOTE

Changes to the BorderIcon and BorderStyle properties aren't reflected at design time. These changes happen at runtime only. This is also the case with other properties, most of which are found on TForm. The reason for this behavior is that it doesn't make sense to change the appearance of certain properties at design time. Take, for example, the Visible property. It's difficult to select a control on the form when its Visible property is set to False because the control is invisible.

Sticky Captions!

You might have noticed that none of the options mentioned allow you to create captionless, resizable forms. Although this isn't impossible, doing so requires a bit of trickery not yet covered. You must override the form's `CreateParams()` method and set the styles required for that window style. The following code snippet does this:

```
unit Nocapu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
public
    { override CreateParams method }
    procedure CreateParams(var Params: TCreateParams); override;
end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);  { Call the inherited Params }
  { Set the style accordingly }
  Params.Style := WS_THICKFRAME or WS_POPUP or WS_BORDER;
end;
end.
```

You'll learn more about the `CreateParams()` method in Chapter 21, "Writing Delphi Custom Components."

You can find an example of a sizable, borderless form in the project `NoCaption.dpr` on the CD-ROM that accompanies this book. This demo also illustrates how to capture the `WM_NCHITTEST` message to enable moving the form without the caption by dragging the form.

Take a look at the `BrdrIcon.dpr` project on the CD-ROM. This project shows how you can change the `BorderIcon` and `BorderStyle` property at runtime so that you see the visual effect. Listing 4.2 shows the main form for this project, which contains the relevant code.

LISTING 4.2 The Main Form for the `BorderStyle/BorderIcon` Project

```
unit MainFrm;

interface
```

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

type
  TMainForm = class(TForm)
    gbBorderIcons: TGroupBox;
    cbSystemMenu: TCheckBox;
    cbMinimize: TCheckBox;
    cbMaximize: TCheckBox;
    rgBorderStyle: TRadioGroup;
    cbHelp: TCheckBox;
    procedure cbMinimizeClick(Sender: TObject);
    procedure rgBorderStyleClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.cbMinimizeClick(Sender: TObject);
var
  IconSet: TBorderIcons;  // Temp variable to hold values.
begin
  IconSet := [];  // Initialize to an empty set
  if cbSystemMenu.Checked then
    IconSet := IconSet + [biSystemMenu]; // Add the biSystemMenu button
  if cbMinimize.Checked then
    IconSet := IconSet + [biMinimize];   // Add the biMinimize button
  if cbMaximize.Checked then
    IconSet := IconSet + [biMaximize];   // Add the biMaximize button
  if cbHelp.Checked then
    IconSet := IconSet + [biHelp];

  BorderIcons := IconSet;                // Assign result to the form's
end;                                     // BorderIcons property.

procedure TMainForm.rgBorderStyleClick(Sender: TObject);
begin
  BorderStyle := TBorderStyle(rgBorderStyle.ItemIndex);
end;

end.
```

NOTE

Some properties in the Object Inspector affect the appearance of your form; others define behavioral aspects for your form. Experiment with each property that's unfamiliar. If you need to know more about a property, use the Delphi 5 help system to find additional information.

Form Reusability: Visual Form Inheritance

A useful feature of Delphi 5 is a concept known as *visual form inheritance*. In the first version of Delphi, you could create a form and save it as a template, but you didn't have the advantage of true *inheritance* (the capability to access the ancestor form's components, methods, and properties). By using inheritance, all descendant forms share the same code as their ancestor. The only overhead involves the methods you add to your descendant forms. Therefore, you also gain the advantage of reducing your application's overall footprint. Another advantage is that changes made to the ancestor code are also applied to its descendants.

The Object Repository

Delphi 5 has a project-management feature that allows programmers to share forms, dialog boxes, data modules, and project templates. This feature is called the *Object Repository*. By using the Object Repository, developers can share the various objects listed with developers of other projects. Additionally, the Object Repository allows developers to maximize code reuse by allowing them to inherit their objects from objects that exist in the Object Repository. Chapter 4 of the Delphi 5 User's Guide covers the Object Repository. It's a good idea to become familiar with this powerful feature.

TIP

In a network environment, you might want to share form templates with other programmers. This is possible by creating a shared repository. In the Environment Options dialog box (select Tools, Environment Options), you can specify the location of a shared repository. Each programmer must map to the same drive that points to this directory location. Then, whenever File, New is selected, Delphi will scan this directory for any shared items in the repository.

Inheriting a form from another form is simple because it's completely built into the Delphi 5 environment. To create a form that descends from another form definition, you simply select File, New from Delphi's main menu, which invokes the New Items dialog box. This dialog box actually gives you a view of the objects that exist in the Object Repository (see the sidebar "The Object Repository"). You then select the Forms page, which lists the forms that have been added to the Object Repository.

NOTE

You don't have to go through the Object Repository to get form inheritance. You can inherit from forms that are in your project. Select File, New and then select the Project page. From there, you can select an existing form in your project. Forms shown in the Project page are not in the Object Repository.

The various forms listed are those that have been added previously to the Object Repository. You'll notice that there are three options for how to include the form in your project: Copy, Inherit, and Use.

Choosing Copy adds an exact duplicate of the form to your project. If the form kept in the Object Repository is modified, this won't affect your copied form.

Choosing Inherit causes a new form class derived from the form you selected to be added to your project. This powerful feature allows you to inherit from the class in the Object Repository so that changes made to the Object Repository's form are reflected by the form in your project as well. This is the option that most developers ought to select.

Choosing Use causes the form to be added to your project as if you had created it as part of the project. Changes you make to the item at design time will appear in all projects that also use the form and any projects that inherit from the form.

The TApplication Class

Every form-based Delphi 5 program contains a global variable, `Application`, of the type `TApplication`. `TApplication` encapsulates your program and performs many behind-the-scenes functions that enable your application to work correctly within the Windows environment. These functions include creating your window class definition, creating the main window for your application, activating your application, processing messages, adding context-sensitive help, processing menu accelerator keys, and handling VCL exceptions.

NOTE

Only form-based Delphi applications contain the global `Application` object. Applications such as console apps don't contain a VCL `Application` object.

You typically won't have to be concerned about the background tasks that `TApplication` performs. However, some situations might necessitate that you delve into the inner workings of `TApplication`.

Because `TApplication` doesn't appear in the Object Inspector, you can't modify its properties there. However, you can choose Project, Options and select the Application page, from which you can set some of the properties for `TApplication`. Mostly, you work with the `TApplication` instance, `Application`, at runtime—that is, you set its property values and assign event handlers to `Application` when the program is running.

TApplication's Properties

`TApplication` has several properties that you can access at runtime. The following sections discuss some of the properties specific to `TApplication` and how you can use them to change the default behavior of `Application` to enhance your project. `TApplication`'s properties are also well documented in the Delphi 5 online help.

The TApplication.ExeName Property

The `ExeName` property of `Application` holds the full path and filename for the project. Because this is a runtime, read-only property, you can't modify it. However, you can read it—or even let your users know where they ran the application from. For example, the following line of code changes a main form's caption to the contents of `ExeName`:

```
Application.MainForm.Caption := Application.ExeName;
```

TIP

Use the `ExtractFileName()` function to retrieve the filename from a string containing the full path of a file:

```
ShowMessage(ExtractFileName(Application.ExeName));
```

Use `ExtractFilePath()` to retrieve the path of a full path string:

```
ShowMessage(ExtractFilePath(Application.ExeName));
```

Finally, use `ExtractFileExt()` to extract the extension of a filename:

```
ShowMessage(ExtractFileExt(Application.ExeName));
```

The TApplication.MainForm Property

In the preceding section, you saw how to access the `MainForm` property to change its `Caption` to reflect the `ExeName` for the application. `MainForm` points to a `TForm` so that you can access any `TForm` property through `MainForm`. You can also access properties that you add to your descendant forms, as long as you typecast `MainForm` accordingly:

```
(MainForm as TForm1).SongTitle := 'The Flood';
```

MainForm is a read-only property. You can specify which form in your application is the main form at design time by using the Forms page in the Project Options dialog box.

The TApplication.Handle Property

The Handle property is an HWND (a *window handle*, in Win32 API terms). The window handle is the owner of all top-level windows in your application. Handle is what makes modal dialog boxes modal over all windows of your application. You don't have to access Handle that often, unless you intend to take over the default behavior of the application in a way that isn't provided by Delphi. You may also refer to the Handle property when using Win32 API functions requiring the application's window handle. We'll discuss Handle more later in the chapter.

The TApplication.Icon and TApplication.Title Properties

The Icon property holds the icon that represents the application when your project is minimized. You can change the application's icon by providing another icon and assigning it to Application.Icon, as described in the later section "Adding Resources to Your Project."

The text that appears next to the icon in the application's task button on the Windows 95/98 taskbar is the application's Title property. If you're running Windows NT, this text appears just underneath the icon. Changing the title of the task button is simple—just make a string assignment to the Title property:

```
Application.Title := 'New Title';
```

Other Properties

The Active property is a read-only Boolean property that indicates whether the application has focus and is active.

The ComponentCount property indicates the number of components that Application contains. Mainly, these components are forms and a THintWindow instance if the Application.ShowHint property is True. ComponentIndex is always -1 for any component that does not have an owner. Therefore, TApplication.ComponentIndex is always -1. This property mainly applies to forms and components on forms.

The Components property is an array of components that belong to the Application. There will be TApplication.ComponentCount items in the Components array. The following code shows how you would add the class names of all components referred to by ComponentCount to a TListBox component:

```
var
  i: integer;
begin
  for i := 0 to Application.ComponentCount - 1 do
    ListBox1.Items.Add(Application.Components[i].ClassName);
end;
```

The HelpFile property contains the Windows help filename, which enables you to add online help to your application. It's used by TApplication.HelpContext and other help invocation methods.

The TApplication.Owner property is always nil because TApplication can't be owned by any other component.

The ShowHint property enables or disables the display of hints for the entire application. The Application.ShowHint property overrides the values of any other component's ShowHint property. Therefore, if Application.ShowHint is False, hints are not displayed for any component.

The Terminated property is True whenever the application has been terminated by closing the main form or by calling the TApplication.Terminate() method.

TApplication's Methods

TApplication has several methods with which you should be familiar. The following sections discuss some of the methods specific to TApplication.

The TApplication.CreateForm() Method

The TApplication.CreateForm() method is defined as follows:

```
procedure CreateForm(InstanceClass: TComponentClass; var Reference)
```

This method creates an instance of a form with the type specified by InstanceClass and assigns that instance to the Reference variable. Earlier, you saw how this method was called in the project's DPR file. The code had the following line, which creates the instance of Form1 of type TForm1:

```
Application.CreateForm(TForm1, Form1);
```

The line would have been created automatically by Delphi 5 if Form1 appeared in the project's Auto-Create list. However, you can call this method elsewhere in your code if you're creating a form that doesn't appear in the Auto-Create list (in which case the form's instance wouldn't have been created automatically). This approach doesn't differ much from calling the form's own Create() method, except that TApplication.CreateForm() checks to see whether the TApplication.MainForm property is nil; if so, CreateForm() assigns the newly created form to Application.MainForm. Subsequent calls to CreateForm() don't affect this assignment. Typically, you don't call CreateForm(); you use a form's Create() method instead.

The TApplication.HandleException() Method

The HandleException() method is where the TApplication instance displays information about exceptions that occur in your project. This information is displayed with a standard exception message box defined by VCL. You can override this message box by attaching an event handler to the `Application.OnException` event, as shown in the later section "Overriding the Application's Exception Handling."

TApplication's HelpCommand(), HelpContext(), and HelpJump() Methods

The `HelpCommand()`, `HelpContext()`, and `HelpJump()` methods each provide a way for you to interface your projects with the Windows help system provided by the `WINHELP.EXE` program that ships with Windows. `HelpCommand()` allows you to call any of the WinHelp macro commands and macros defined in your help file. `HelpContext()` allows you to launch a help page in the help file specified by the `TApplication.HelpFile` property. The page displayed is based on the value of the `Context` parameter passed to `HelpContext()`. `HelpJump()` is much like `HelpContext()`, except that it takes a `JumpID` string parameter.

The TApplication.ProcessMessages() Method

`ProcessMessages()` causes your application to actively go get any messages that are waiting for it and then process them. This is useful when you have to perform a process within a tight loop and you don't want your code to prevent you from executing other code (such as processing an abort button). In contrast, `TApplication.HandleMessages()` puts the application into an idle state if there are no messages, whereas `ProcessMessages()` won't put it in an idle state. The `ProcessMessages()` method is used in Chapter 10, "Printing in Delphi 5."

The TApplication.Run() Method

Delphi 5 automatically places the `Run()` method within the project file's main block. You never have to call this method yourself, but you should know where it goes and what it does in case you ever have to modify the project file. Basically, `TApplication.Run()` first sets up an exit procedure for the project, which ensures that all components are freed when the project ends. It then enters a loop that calls the methods to process messages for the project until the application is terminated.

The TApplication.ShowException() Method

The `ShowException()` method simply takes an exception class as a parameter and displays a message box with information about that exception. This method comes in handy if you're overriding the `Application`'s exception handling method, as shown in the later section "Overriding the Application's Exception Handling."

Other Methods

`TApplication.Create()` creates the `TApplication` instance. This method is called internally by Delphi 5; you should never call it.

`TApplication.Destroy()` destroys the `TApplication` instance. This method is called internally by Delphi 5. You should never call this method.

`TApplication.MessageBox()` allows you to display a Windows message box. However, this method doesn't require that you pass a window's handle, as the Windows `MessageBox()` function does.

`TApplication.Minimize()` places your application in a minimized state.

`TApplication.Restore()` restores your application to its previous size from a minimized or maximized state.

`TApplication.Terminate()` terminates the execution of your application. `Terminate` is an indirect call to `PostQuitMessage`, resulting in a graceful shutdown of the application (unlike `Halt()`).

NOTE

Use the `TApplication.Terminate()` method to halt an application. `Terminate()` calls the Windows API function `PostQuitMessage()`, which posts a message to your application's message queue. VCL responds by properly freeing objects that have been created in your application. The `Terminate()` method is a clean way to stop your application's process. It's important to note that your application does not terminate at the call to `Terminate()`. Instead, it continues to run until the application returns to its message queue and retrieves the `WM_QUIT` message. `Halt()`, on the other hand, forcibly terminates your application without freeing any objects or shutting down gracefully. Execution does not return from a call to `Halt()`.

TApplication's Events

`TApplication` has several events to which you can add event handlers. In past versions of Delphi, these events were not accessible from the Object Inspector (for example, the events for the form or components on the Component Palette). You had to add an event handler to the `Application` variable by first defining the handler as a method and then assigning that method to the handler at runtime. Delphi 5 adds a new component to the Additional page of the Component Palette—`TApplicationEvents`. This component allows you to assign event handlers at design time to the global `Application` instance. Table 4.2 lists the events associated with `TApplication`.

TABLE 4.2 `TApplication` and `TApplicationEvents` Events

Event	*Description*
`OnActivate`	Occurs when the application becomes active; `OnDeactivate` occurs when the application stops being active (for example, when you switch to another application).

Event	*Description*
`OnException`	Occurs when an unhandled exception has occurred; you can add default processing for unhandled exceptions. `OnException` occurs if the exception makes it all the way up to the application object. Normally, you should allow exceptions to be handled by the default exception handler and not trapped by `Application.OnException` or lower code. If you must trap an exception, reraise it and make sure that the exception instance carries a full description of the situation so that the default exception handler can present useful information.
`OnHelp`	Occurs for any invocation of the help system, such as when F1 is pressed or when the following methods are called: `HelpCommand()`, `HelpContext()`, and `HelpJump()`.
`OnMessage`	Enables you to process messages before they're dispatched to their intended controls. `OnMessage` gets to peek at all messages posted to all controls in the application. Exercise caution when using `OnMessage` because it could result in a bottleneck.
`OnHint`	Enables you to display hints associated with controls when the mouse is positioned over the control. An example of this is a status line hint.
`OnIdle`	Occurs when the application is switched into an idle state. `OnIdle` is not called continuously. Once in the idle state, an application will not wake up until it receives a message.

You work more with `TApplication` later in this chapter as well as in other projects in other chapters.

NOTE

The `TApplication.OnIdle` event provides a handy way to perform certain processing when no user interaction is occurring. One common use for the `OnIdle` event handler is to update menus and speedbuttons based on the status of the application.

The `TScreen` Class

The `TScreen` class simply encapsulates the state of the screen on which your applications runs. `TScreen` is not a component that you add to your Delphi 5 forms, nor do you create it dynamically during runtime. Delphi 5 automatically creates a `TScreen` global variable called `Screen`, which you can access from within your application. The `TScreen` class contains several properties that you'll find useful. These properties are listed in Table 4.3.

TABLE 4.3 `TScreen` Properties

Property	*Meaning*
`ActiveControl`	A read-only property that indicates which control on the screen has current focus. As focus shifts from one control to another, `ActiveControl` is assigned the newly focused control before the `OnExit` event of the control losing focus finishes.
`ActiveForm`	Indicates the form that has focus. This property is set when another form switches focus or when the Delphi 5 application gains focus from another application.
`Cursor`	The cursor shape that's global to the application. By default, this is set to `crDefault`. Each windowed component has its own `Cursor` property that may be modified. However, when the cursor is set to something other than `crDefault`, all other controls reflect that change until `Screen.Cursor` is set back to `crDefault`. Another way to look at this is `Screen.Cursor = crDefault` means "ask the control under the mouse what cursor should be displayed." `Screen.Cursor <> crDefault` means "don't ask."
`Cursors`	A list of all cursors available to the screen device.
`DataModules`	A list of all data modules belonging to the application.
`DataModuleCount`	The number of data modules belonging to the application.
`FormCount`	The number of available forms in the application.
`Forms`	A list of forms available to the application.
`Fonts`	A list of font names available to the screen device.
`Height`	The height of the screen device in pixels.
`PixelsPerInch`	Indicates the relative scale of the system font.
`Width`	The width of the screen device in pixels.

Defining a Common Architecture: Using the Object Repository

Delphi makes it so easy to develop applications that you can get 60 percent into your application development before you realize that you should have spent more time up front on application architecture. A common problem with development is that developers are too anxious to get coding before spending the appropriate time really thinking about application design. This alone is one of the biggest contributors to project failure.

Thoughts on Application Architecture

This is not a book on architecture or object-oriented analysis and design. However, we strongly feel that this is one of the most important aspects of application development in addition to requirements, detail design, and everything else that constitutes the initial 80 percent of a product before coding begins. We've listed some of our favorite references on topics such as object-oriented analysis in Appendix C, "Suggested Reading." It would be to your best interest to research this topic thoroughly before you roll your sleeves up and start coding.

Here are a few examples of the many issues that come into play when considering application architecture:

- Does the architecture support code reuse?
- Is the system organized so that modules, objects, and so on are localized?
- Can changes more easily be made to the architecture?
- Are the user interface and back end localized so that either can be replaced?
- Does the architecture support a team development effort? In other words, can team members easily work on separate modules without overlap?

These are just a few of the things to consider during development.

Volumes have been written on this topic alone, so we won't attempt to compete with that information. We do, however, hope that we've sparked your interest enough to make you learn about it if you aren't already an architecture guru. The following sections illustrate a simple method of architecting a common UI for a database application and how Delphi can help you do that.

Delphi's Inherent Architecture

You'll often hear that you don't have to be a component writer to be a Delphi developer. Although true, it's also true that if you're a component writer, you're a much better Delphi developer.

This is because component writers clearly understand the object model and architecture that Delphi applications inherit just by being Delphi applications. This means that component writers are better equipped to take advantage of this powerful and flexible model in their own applications. In fact, you've probably already heard that Delphi is written in Delphi. Delphi is an example of an application written with the same inherent architecture that your applications can also use.

Even if you don't intend to write components, you'll be better off if you learn it anyway. Become thoroughly knowledgeable of the VCL and the Object Pascal model as well as of the Win32 operating system.

An Architecture Example

To demonstrate the power of form inheritance as well as the use of the Object Repository, we're going to define a common application architecture. The issues we're focusing on are code reusability, flexibility for change, consistency, and facility for team development.

The form class hierarchy, or rather, *framework*, consists of forms to be used specifically for database applications. These forms are typical of most database applications. The forms should be aware of the state of the database operation (edit, add, or browse). They should also contain the common controls used in performing these operations on a database table, such as a toolbar and status bar whose displays and controls change according to the form's state. Additionally, they should provide an event that can be invoked whenever the form mode changes.

This framework should also enable a team to work on isolated parts of an application without requiring the entire application's source code. Otherwise, there's the likelihood that different programmers would modify the same files.

For now, this framework's hierarchy will contain three levels. This will be expanded on later in the book.

Table 4.4 describes the purpose of each form in the framework.

TABLE 4.4 Database Form Framework

Form Class	*Purpose*
`TChildForm = class(TForm)`	Provides the capability to be inserted as a child to another window
`TDBModeForm = class(TChildForm)`	Is aware of a database state (browse, insert, edit) and contains an event to be invoked upon state change
`TDBNavStatForm = class(TDBBaseForm)`	Typical database entry form that's aware of its state and contains the standard navigation bar and status bar to be used by all database applications

The Child Form (TChildForm)

`TChildForm` is a base class for forms that can be launched as independent modal and modeless forms and can become child windows to any other window.

This capability makes it easy for a team of developers to work on separate pieces of an application apart from the overall application. It also provides a nice UI feature in that the user can launch a form as a separate entity in an application, even though that might not be the normal method of interacting with that form. Listing 4.3 is the source for `TChildForm`. You'll find this and all the other forms to be placed in the Object Repository in the `\Code` directory on the CD-ROM.

LISTING 4.3 `TChildForm` Source

```
unit ChildFrm;

interface

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

type

  TChildForm = class(TForm)
  private
    FAsChild: Boolean;
    FTempParent: TWinControl;
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure Loaded; override;
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(AOwner: TComponent;
        AParent: TWinControl); reintroduce; overload;

    // The method below must be overridden to return either the main menu
    // of the form, or nil.
    function GetFormMenu: TMainMenu; virtual; abstract;
    function CanChange: Boolean; virtual;
  end;

implementation

{$R *.DFM}
constructor TChildForm.Create(AOwner: TComponent);
begin
  FAsChild := False;
  inherited Create(AOwner);
end;

constructor TChildForm.Create(AOwner: TComponent; AParent: TWinControl);
begin
```

```
  FAsChild := True;
  FTempParent := aParent;
  inherited Create(AOwner);
end;

procedure TChildForm.Loaded;
begin
  inherited;
  if FAsChild then
  begin
    align := alClient;
    BorderStyle := bsNone;
    BorderIcons := [];
    Parent := FTempParent;
    Position := poDefault;
  end;
end;

procedure TChildForm.CreateParams(var Params: TCreateParams);
Begin
  Inherited CreateParams(Params);
  if FAsChild then
    Params.Style := Params.Style or WS_CHILD;
end;

function TChildForm.CanChange: Boolean;
begin
  Result := True;
end;

end.
```

This listing demonstrates a couple of techniques. First, it shows how to use the overload extensions to the Object Pascal language, and second, it shows how to make a form a child of another window.

Providing a Second Constructor

You'll notice that we've declared two constructors for this child form. The first constructor declared is used when the form is created as a normal form. This is the constructor with one parameter. The second constructor, which takes two parameters, is declared as an overloaded constructor. You would use this constructor to create the form as a child window. The parent to the form gets passed as the `AParent` parameter. Notice that we've used the `reintroduce` directive to suppress the warning about hiding the virtual constructor.

The first constructor simply sets the FAsChild variable to False to ensure that the form is created normally. The second constructor sets the value to True and sets FTempParent to the AParent parameter. This value is used later as the parent of the child form in the Loaded() method.

Making a Form a Child Window

To make a form a child window, there are a few things you need to do. First, you have to make sure that various property settings have been set, which you'll see is done programmatically in TChildForm.Loaded(). In Listing 4.3, we ensure that when the form becomes a child it doesn't look like a dialog box. We do this by removing the border and any border icons. We also make sure that the form is client-aligned and set the parent to the window referred to by the FTempParent variable. If this form were going to be used as a child only, we could have made these settings at design time. However, this form will also be launched as a normal form, so we set these properties only if the FAsChild variable is True.

We also have to override the CreateParams() method to tell Windows to create the form as a child window. We do this by setting the WS_CHILD style in the Params.Style property.

This base form is not restricted to a database application. In fact, you can use it for any form that you want to have child window capabilities. You'll find a demo of this child form being used as both a normal form and as a child form in the ChildTest.dpr project found in the \Form Framework directory on the CD-ROM.

NOTE

Delphi 5 introduces frames to the VCL. Frames work so that they can be embedded within a form. Because frames serve as containers for components, they function much like the child form shown previously. We'll discuss frames in more detail momentarily.

The Database Base Mode Form (TDBModeForm)

TDBModeForm is a descendant of TChildForm. Its purpose is to be aware of the state of a table (browse, insert, and edit). This form also provides an event that occurs whenever the mode is changed.

Listing 4.4. shows the source code for TDBModeForm.

LISTING 4.4 TDBModeForm

```
unit DBModeFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  CHILDFRM;

type

  TFormMode = (fmBrowse, fmInsert, fmEdit);

  TDBModeForm = class(TChildForm)
  private
    FFormMode      : TFormMode;
    FOnSetFormMode : TNotifyEvent;
  protected
    procedure SetFormMode(AValue: TFormMode); virtual;
    function  GetFormMode: TFormMode; virtual;
  public
    property FormMode: TFormMode read GetFormMode write SetFormMode;
  published
    property OnSetFormMode: TNotifyEvent read FOnSetFormMode
       write FOnSetFormMode;

  end;

var
  DBModeForm: TDBModeForm;

implementation

{$R *.DFM}

procedure TDBModeForm.SetFormMode(AValue: TFormMode);
begin
  FFormMode := AValue;
  if Assigned(FOnSetFormMode) then
    FOnSetFormMode(self);
end;

function TDBModeForm.GetFormMode: TFormMode;
begin
  Result := FFormMode;
end;

end.
```

The implementation of `TDBModeForm` is straightforward. Although we're using some techniques we haven't yet discussed, you should be able to follow what's happening here. First, we just defined the enumerated type, `TFormMode`, to represent the form's state. Then we provided the `FormMode` property and its read and write methods. The technique for creating the property and read/write methods is discussed further in Chapter 21, "Writing Delphi Custom Components."

A demo using `TDBModeForm` is in the project `FormModeTest.DPR` found in the `\Form Framework` directory on the CD-ROM.

The Database Navigation/Status Form (TDBNavStatForm)

`TDBNavStatForm` brings the bulk of the functionality of this framework. This form contains the common set of components to be used in our database applications. In particular, it has a navigation bar and status bar that automatically change based on the form's state. For example, you'll see that the Accept and Cancel buttons are initially disabled when the form is in the state of `fsBrowse`. However, when the user places the form in the `fsInsert` or `fsEdit` state, the buttons become enabled. The status bar also displays the state the form is in.

Listing 4.5 shows the source code for `TDBNavStatForm`. Notice that we've eliminated the component list from the listing. You'll see these if you load the demo project for this form.

LISTING 4.5 `TDBNavStatForm`

```
unit DBNavStatFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBMODEFRM, ComCtrls, ToolWin, Menus, ExtCtrls, ImgList;

type
  TDBNavStatForm = class(TDBModeForm)
    { components not included in listing. }
    procedure sbAcceptClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
  private
    { Private declarations }
  protected
    procedure Setbuttons; virtual;
    procedure SetStatusBar; virtual;
    procedure SetFormMode(AValue: TFormMode); override;
  public
```

continues

LISTING 4.5 Continued

```
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(AOwner: TComponent; AParent: TWinControl); overload;
    procedure SetToolBarParent(AParent: TWinControl);
    procedure SetStatusBarParent(AParent: TWinControl);
  end;

var
  DBNavStatForm: TDBNavStatForm;

implementation

{$R *.DFM}

{ TDBModeForm3 }

procedure TDBNavStatForm.SetFormMode(AValue: TFormMode);
begin
  inherited SetFormMode(AValue);
  SetButtons;
  SetStatusBar;
end;

procedure TDBNavStatForm.Setbuttons;

  procedure SetBrowseButtons;
  begin
    sbAccept.Enabled  := False;
    sbCancel.Enabled  := False;

    sbInsert.Enabled  := True;
    sbDelete.Enabled  := True;
    sbEdit.Enabled    := True;

    sbFind.Enabled    := True;
    sbBrowse.Enabled  := True;


    sbFirst.Enabled   := True ;
    sbPrev.Enabled    := True ;
    sbNext.Enabled    := True ;
    sbLast.Enabled    := True ;
  end;

  procedure SetInsertButtons;
  begin
    sbAccept.Enabled  := True;
    sbCancel.Enabled  := True;

    sbInsert.Enabled  := False;
    sbDelete.Enabled  := False;
    sbEdit.Enabled    := False;

    sbFind.Enabled    := False;
    sbBrowse.Enabled  := False;

    sbFirst.Enabled   := False;
    sbPrev.Enabled    := False;
    sbNext.Enabled    := False;
    sbLast.Enabled    := False;
  end;

  procedure SetEditButtons;
  begin
    sbAccept.Enabled  := True;
    sbCancel.Enabled  := True;

    sbInsert.Enabled  := False;
    sbDelete.Enabled  := False;
    sbEdit.Enabled    := False;

    sbFind.Enabled    := False;
    sbBrowse.Enabled  := True;

    sbFirst.Enabled   := False;
    sbPrev.Enabled    := False;
    sbNext.Enabled    := False;
    sbLast.Enabled    := False;
  end;

begin
  case FormMode of
    fmBrowse: SetBrowseButtons;
    fmInsert: SetInsertButtons;
    fmEdit:   SetEditButtons;
  end; { case }

end;

procedure TDBNavStatForm.SetStatusBar;
begin
```

continues

LISTING 4.5 Continued

```
  case FormMode of
    fmBrowse: stbStatusBar.Panels[1].Text := 'Browsing';
    fmInsert: stbStatusBar.Panels[1].Text := 'Inserting';
    fmEdit:   stbStatusBar.Panels[1].Text := 'Edit';
  end;

  mmiInsert.Enabled := sbInsert.Enabled;
  mmiEdit.Enabled   := sbEdit.Enabled;
  mmiDelete.Enabled := sbDelete.Enabled;
  mmiCancel.Enabled := sbCancel.Enabled;
  mmiFind.Enabled   := sbFind.Enabled;

  mmiNext.Enabled   := sbNext.Enabled;
  mmiPrevious.Enabled := sbPrev.Enabled;
  mmiFirst.Enabled  := sbFirst.Enabled;
  mmiLast.Enabled   := sbLast.Enabled;

end;

procedure TDBNavStatForm.sbAcceptClick(Sender: TObject);
begin
  inherited;
   FormMode := fmBrowse;
end;

procedure TDBNavStatForm.sbInsertClick(Sender: TObject);
begin
  inherited;
  FormMode := fmInsert;
end;

procedure TDBNavStatForm.sbEditClick(Sender: TObject);
begin
  inherited;
  FormMode := fmEdit;
end;

constructor TDBNavStatForm.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FormMode := fmBrowse;
end;

constructor TDBNavStatForm.Create(AOwner: TComponent; AParent: TWinControl);
```

```
begin
  inherited Create(AOwner, AParent);
  FormMode := fmBrowse;
end;

procedure TDBNavStatForm.SetStatusBarParent(AParent: TWinControl);
begin
  stbStatusBar.Parent := AParent;
end;

procedure TDBNavStatForm.SetToolBarParent(AParent: TWinControl);
begin
  tlbNavigationBar.Parent := AParent;
end;

end.
```

The event handlers for the various `TToolButton` components basically set the form to its appropriate state. This, in turn, invokes the `SetFormMode()` methods, which we've overridden to call the `SetButtons()` and `SetStatusBar()` methods. `SetButtons()` enables or disables the buttons accordingly based on the form's mode.

You'll notice that we've also provided two procedures to change the parent of the `TToolBar` and `TStatusBar` components on the form. This functionality is provided so that when the form is invoked as a child window, we can set the parent of these components to the main form. When you run the demo provided in the `\Form Framework` directory on the CD-ROM, you'll see why this makes sense.

As stated earlier, `TDBNavStatForm` inherits the functionality to be an independent form as well as a child window. The demo invokes an instance of `TDBNavStatForm` with the following code:

```
procedure TMainForm.btnNormalClick(Sender: TObject);
var
  LocalNavStatForm: TNavStatForm;
begin
  LocalNavStatForm := TNavStatForm.Create(Application);
  try
    LocalNavStatForm.ShowModal;
  finally
    LocalNavStatForm.Free;
  end;
end;
```

The following code shows how to invoke the form as a child window:

```
procedure TMainForm.btnAsChildClick(Sender: TObject);
begin
```

```
  if not Assigned(FNavStatForm) then
  begin
    FNavStatForm := TNavStatForm.Create(Application, pnlParent);
    FNavStatForm.SetToolBarParent(self);
    FNavStatForm.SetStatusBarParent(self);
    mmMainMenu.Merge(FNavStatForm.mmFormMenu);
    FNavStatForm.Show;
    pnlParent.Height := pnlParent.Height - 1;
  end;
end;
```

This code not only invokes the form as a child to the `TPanel` component, `pnlParent`, but also sets the form's `TToolBar` and `TStatusBar` components to reside on the main form. Additionally, notice the call to `TMainForm.mmMainMenu.Merge()`. This allows us to merge any menus that reside on the `TDBNavStatForm` instance with `MainForm`'s main menu. Naturally, when we free the `TDBNavStatForm` instance, we must also make a call to `TMainForm.mmMainMenu.UnMerge()`, as shown in the following code:

```
procedure TMainForm.btnFreeChildClick(Sender: TObject);
begin
  if Assigned(FNavStatForm) then
  begin
    mmMainMenu.UnMerge(FNavStatForm.mmFormMenu);
    FNavStatForm.Free;
    FNavStatForm := nil;
  end;
end;
```

Take a look at the demo provided on the CD-ROM. Figure 4.1 shows this project with both the child form and independent `TDBNavStatForm` instances created. Notice that we've placed a `TImage` component on the form to better display the form as a child. Figure 4.1 shows how we use the same child form (the one with the picture) as both an embedded window and as a separate form.

Later, we'll use and expand on this same framework to create a fully functional database application.

Using Frames in Application Framework Design

Delphi 5 now has frames. They allow you to create component containers that may be embedded within another form. This is similar to what we've already demonstrated using `TChildForm`. Frames, however, allow you to manipulate your component containers at design time and to add them to the Component Palette so that they may be reused. Listing 4.6 shows the main form for a project similar to the child form demo, except that it uses frames.

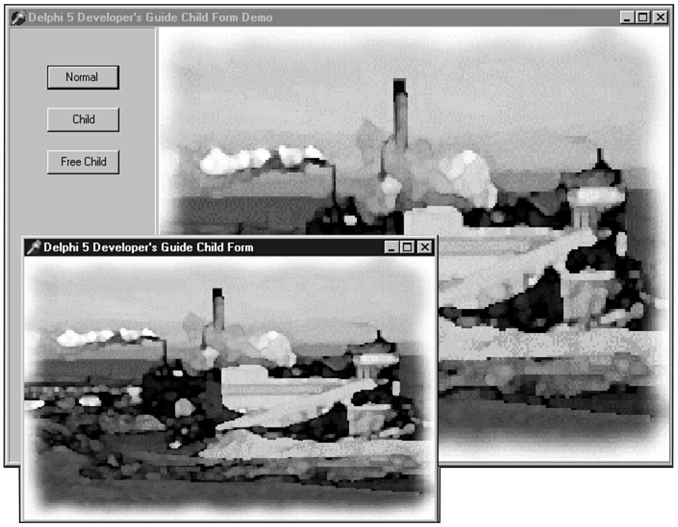
Delphi 5 Developer's Guide Child Form Demo
Normal
Child
Free Child
Delphi 5 Developer's Guide Child Form

FIGURE 4.1
`TDBNavStatForm` *as a normal form and as a child window.*

LISTING 4.6 Frames Demo

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    spltrMain: TSplitter;
    pnlParent: TPanel;
    pnlMain: TPanel;
    btnFrame1: TButton;
    btnFrame2: TButton;
    procedure btnFrame1Click(Sender: TObject);
    procedure btnFrame2Click(Sender: TObject);
  private
    { Private declarations }
    FFrame: TFrame;
  public
```

continues

LISTING 4.6 Continued

```
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses Frame1Fram, Frame2Fram;

{$R *.DFM}

procedure TMainForm.btnFrame1Click(Sender: TObject);
begin
  if FFrame <> nil then
    FFrame.Free;
  FFrame := TFrame1.Create(pnlParent);
  FFrame.Align := alClient;
  FFrame.Parent := pnlParent;
end;

procedure TMainForm.btnFrame2Click(Sender: TObject);
begin
  if FFrame <> nil then
    FFrame.Free;
  FFrame := TFrame2.Create(pnlParent);
  FFrame.Align := alClient;
  FFrame.Parent := pnlParent;
end;

end.
```

In Listing 4.6, we show a main form that contains two panes made up of two separate panels. The panel on the right will serve to hold our frame. We've defined two separate frames. The private field, `FFrame`, is a reference to a `TFrame` class. Since, both our frames descend directly from `TFrame`, `FFrame` can refer to both our `TFrame` descendants. The two buttons on the main form each create a different `TFrame` and assign it to `FFrame`. The effect is the same as with `TChildForm`. The demo `FrameDemo.dpr` is located on the accompanying CD-ROM.

Miscellaneous Project Management Routines

The projects that follow are a series of project-management routines that have been helpful to many Delphi 5 developers.

Adding Resources to Your Project

Earlier, you learned that the RES file is the resource file for your application. You also learned what Windows resources are. You can add resources to your applications by creating a separate RES file to store your bitmaps, icons, cursors, and so on.

You must use a resource editor to build an RES file. After you create your RES file, you simply link it to your application by placing this statement in the application's DPR file:

```
{$R MYFILE.RES}
```

This statement can be placed directly under the following statement, which links the resource file with the same name as the project file to your project:

```
{$R *.RES}
```

If you've done this correctly, you can then load resources from the RES file by using the `TBitmap.LoadFromResourceName()` or `TBitmap.LoadFromResourceID()` method. Listing 4.7 shows the technique for loading a bitmap, icon, and cursor from a resource (RES) file. You can find this project, `Resource.dpr`, on the CD-ROM that accompanies this book. Notice that the API functions used here—`LoadIcon()` and `LoadCursor()`—are all documented in the Windows API help.

NOTE

The Windows API provides a function called `LoadBitmap()` that loads a bitmap (as its name implies). However, this function does not return a color palette and therefore does not work for loading 256-color bitmaps. Use `TBitmap.LoadFromResouceName()` or `TBitmap.LoadFromResouceID()` instead.

LISTING 4.7 Examples of Loading Resources from an RES File

```
unit MainFrm;
interface
uses
  Windows, Forms, Controls, Classes, StdCtrls, ExtCtrls;

const
  crXHair = 1; // Declare a constant for the new cursor. This value
type           // must be a positive number. or less than -20.

  TMainForm = class(TForm)
    imgBitmap: TImage;
    btnChemicals: TButton;
```

continues

Listing 4.7 Continued

```
    btnClear: TButton;
    btnChangeIcon: TButton;
    btnNewCursor: TButton;
    btnOldCursor: TButton;
    btnOldIcon: TButton;
    btnAthena: TButton;
    procedure btnChemicalsClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
    procedure btnChangeIconClick(Sender: TObject);
    procedure btnNewCursorClick(Sender: TObject);
    procedure btnOldCursorClick(Sender: TObject);
    procedure btnOldIconClick(Sender: TObject);
    procedure btnAthenaClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnChemicalsClick(Sender: TObject);
begin
  { Load the bitmap from the resource file. The bitmap must be
    specified in all CAPS! }
  imgBitmap.Picture.Bitmap.LoadFromResourceName(hInstance, 'CHEMICAL');
end;

procedure TMainForm.btnClearClick(Sender: TObject);
begin
  imgBitmap.Picture.Assign(nil); // Clear the image
end;

procedure TMainForm.btnChangeIconClick(Sender: TObject);
begin
  { Load the icon from the resource file. The icon must be
    specified in all CAPS! }
  Application.Icon.Handle := LoadIcon(hInstance, 'SKYLINE');
end;

procedure TMainForm.btnNewCursorClick(Sender: TObject);
begin
  { Assign the new cursor to the Screen's Cursor array }
  Screen.Cursors[crXHair] := LoadCursor(hInstance, 'XHAIR');
  Screen.Cursor := crXHair;  // Now change the cursor
```

```
end;

procedure TMainForm.btnOldCursorClick(Sender: TObject);
begin
  // Change back to default cursor
  Screen.Cursor := crDefault;
end;

procedure TMainForm.btnOldIconClick(Sender: TObject);
begin
  { Load the icon from the resource file. The icon must be
    specified in all CAPS! }
  Application.Icon.Handle := LoadIcon(hInstance, 'DELPHI');
end;

procedure TMainForm.btnAthenaClick(Sender: TObject);
begin
  { Load the bitmap from the resource file. The bitmap must be
    specified in all CAPS! }
  imgBitmap.Picture.Bitmap.LoadFromResourceName(hInstance, 'ATHENA');
end;

end.
```

Changing the Screen's Cursor

Probably one of the most commonly used `TScreen` properties is the `Cursor` property, which enables you to change the global cursor for the application. For example, the following code changes the current cursor to an hourglass to indicate that users must wait while a lengthy process executes:

```
Screen.Cursor := crHourGlass
{ Do some lengthy process }
Screen.Cursor := crDefault;
```

`crHourGlass` is a predefined constant that indexes into the `Cursors` array. There are other cursor constants, such as `crBeam` and `crSize`. The existing cursor values range from `0` to `-20` (`crDefault` to `crHelp`). Look in the online help for the `Cursors` property to see a list of all available cursors. You can assign these values to `Screen.Cursor` when necessary.

You also can create your own cursors and add them to the `Cursors` property array. To do this, you must first define a constant with a value that doesn't conflict with the already-available cursors. Predefined cursor values are from `-20` to `0`. Application cursors should only use positive ID numbers. All negative cursor ID numbers are reserved by Borland. Here's an example:

```
crCrossHair := 1;
```

You can use any resource editor (such as the Image Editor that ships with Delphi 5) to create your custom cursor. You must save the cursor into a resource (RES) file. One important point: You must give your RES file a different name than that of your project. Remember that Delphi 5 creates a RES file of the same name as your project whenever you compile your project. You don't want Delphi 5 to overwrite the cursor you create. When you compile your project, make sure that the RES file is in the same directory as your source files so that Delphi 5 will link the cursor resource with your application. You tell Delphi 5 to link the RES file by placing a statement such as the following into the application's DPR file:

```
{$R CrossHairRes.RES}
```

Finally, you must add the following lines of code to load the cursor, add it to the `Cursors` property, and then switch to that cursor:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crCrossHair] := LoadCursor (hInstance, 'CROSSHAIR');
  Screen.Cursor := crCrossHair;
end;
```

Here you use the `LoadCursor()` Win32 API function to load the cursor. `LoadCursor()` takes two parameters: An instance handle to the module from which you want to get the cursor and the name of the cursor as specified in the RES file. Make sure to write the cursor name in the file in ALL CAPS!

`hInstance` refers to the application currently running. Next, assign the value returned from `LoadCursor()` to the `Cursors` property at the location specified by `crCrossHair`, which was previously defined. Finally, assign the current cursor to `Screen.Cursor`.

For an example, locate the project `CrossHair.dpr` on the CD-ROM. This project loads and changes to the crosshair cursor created here and placed in the file `CrossHairRes.res`.

You might also want to invoke the Image Editor by selecting Tools, Image Editor and opening the `CrossHairRes.res` file to see how the cursor was created.

Preventing Multiple Instances of a Form from Being Created

If you use `Application.CreateForm()` or `TForm.Create()` in your code to create a form instance, it's a good idea to ensure that no instance of the form is being held by the `Reference` parameter (as described in the earlier section "The `TForm` Class"). The following code fragment shows this:

```
begin
  if not Assigned(SomeForm) then begin
    Application.CreateForm(TSomeForm, SomeForm);
    try
      SomeForm.ShowModal;
    finally
      SomeForm.Free;
      SomeForm := nil;
    end;
  end
  else
    SomeForm.ShowModal;
end;
```

In this code, it's necessary to assign `nil` to the `SomeForm` variable after it has been destroyed. Otherwise, the `Assigned()` method doesn't function properly, and the method fails. This wouldn't work for a modeless form, however. With modeless forms, you can't determine in code when the form is going to be destroyed. Therefore, you must make the `nil` assignment from within the `OnDestroy` event handler of the form being destroyed. This method was described earlier in this chapter.

Adding Code to the DPR File

You can place code in the project's DPR file before you launch your main form. Such code can be initialization code, a splash screen, database initialization—anything you deem necessary before the main form is displayed. You also have the opportunity to terminate the application before the main form comes up. Listing 4.8 shows a DPR file that prompts the user for a password before granting access to the application. This project is also on the CD-ROM as `Initialize.dpr`.

LISTING 4.8 The `Initialize.dpr` File, Showing Project Initialization

```
program Initialize;

uses
  Forms,
  Dialogs,
  Controls,
  MainFrm in 'MainFrm.pas' {MainForm};

{$R *.RES}

var
  Password: String;
begin
  if InputQuery('Password', 'Enter your password', PassWord) then
    if Password = 'D5DG' then
```

continues

LISTING 4.8 Continued

```
    begin
      // Other initialization routines can go here.
      Application.CreateForm(TMainForm, MainForm);
    Application.Run;
    end
    else
      MessageDlg('Incorrect Password, terminating program', mtError, [mbok],
0);
end.
```

Overriding the Application's Exception Handling

The Win32 system has a powerful error-handling capability—exceptions. By default, whenever an exception occurs in your project, the `Application` instance automatically handles that exception by displaying to the user a standard error box.

As you build larger applications, you'll start to define exception classes of your own. Perhaps the Delphi 5 default exception handling will no longer suit your needs because you have to perform special processing on a specific exception. In such cases, it will be necessary to override `TApplication`'s default exception handling and replace it with your own custom routine.

You saw that `TApplication` has an `OnException` event handler to which you can add code. When an exception occurs, this event handler is called. There you can perform your special processing so that the default exception message doesn't show.

However, recall that the `TApplication` object's properties aren't editable from the Object Inspector. Therefore, you must use the `TApplicationEvents` component to add specialized exception handling to your application.

Listing 4.9 shows you what you need to do to override the application's default exception handling.

LISTING 4.9 Main Form for the Exception Override Demo

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, AppEvnts, Buttons;

type
```

```
  ENotSoBadError = class(Exception);
  EBadError      = class(Exception);
  ERealBadError  = class(Exception);

  TMainForm = class(TForm)
    btnNotSoBad: TButton;
    btnBad: TButton;
    btnRealBad: TButton;
    appevnMain: TApplicationEvents;
    procedure btnNotSoBadClick(Sender: TObject);
    procedure btnBadClick(Sender: TObject);
    procedure btnRealBadClick(Sender: TObject);
    procedure appevnMainException(Sender: TObject; E: Exception);
  public
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnNotSoBadClick(Sender: TObject);
begin
  raise ENotSoBadError.Create('This isn''t so bad!');
end;

procedure TMainForm.btnBadClick(Sender: TObject);
begin
  raise EBadError.Create('This is bad!');
end;

procedure TMainForm.btnRealBadClick(Sender: TObject);
begin
  raise ERealBadError.Create('This is real bad!');
end;

procedure TMainForm.appevnMainException(Sender: TObject; E: Exception);
var
  rslt: Boolean;
begin
  if E is EBadError then
  begin
  { Show a custom message box and prompt for application termination. }
    rslt := MessageDlg(Format('%s %s %s %s %s', ['An', E.ClassName,
```

continues

LISTING 4.9 Continued

```
        'exception has occurred.', E.Message, 'Quit App?']),
         mtError, [mbYes, mbNo], 0) = mrYes;
    if rslt then
       Application.Terminate;
  end
  else if E is ERealBadError then
  begin // Show a custom message
        // and terminate the application.
    MessageDlg(Format('%s %s %s %s %s', ['An', E.ClassName,
        'exception has occured.', E.Message, 'Quitting Application']),
         mtError, [mbOK], 0);
    Application.Terminate;
  end
  else // Perform default exception handling
    Application.ShowException(E);
end;

end.
```

In Listing 4.9, the `appevnMainException()` method is the `OnException` event handler to the `TApplicationEvent` component. This event handler uses RTTI to check the type of exception that occurred and performs special processing based on the exception type. The comments in the code discuss the process. You'll also find the project that uses these routines, `OnException.dpr`, on the CD-ROM accompanying this book.

TIP

If the Stop on Delphi Exceptions check box is selected in the Language Exceptions page of the Debugger Options dialog box (accessed by selecting Tools, Debugger Options), Delphi 5's IDE debugger reports the exception in its own dialog box, before your application has a chance to handle the exception. Although useful for debugging, having this check box selected can be annoying when you want to see how your project handles exceptions. Disable the option to make your project run normally.

Displaying a Splash Screen

Suppose you want to create a splash screen for your project. This form can display when you launch your application and can stay visible while your application initializes. Displaying a splash screen is actually simple. Here are the initial steps for creating a splash screen:

1. After creating your application's main form, create another form to represent the splash screen. Call this form `SplashForm`.
2. Use the Project, Options menu to ensure that `SplashForm` is not in the Auto-Create list.
3. Assign `bsNone` to `SplashForm`'s `BorderStyle` property and `[ ]` to its `BorderIcons` property.
4. Place a `TImage` component onto `SplashForm` and assign `alClient` to the image's `Align` property.
5. Load a bitmap into the `TImage` component by selecting its `Picture` property.

Now that you've designed the splash screen, you only have to edit the project's DPR file to display it. Listing 4.10 shows the project file (DPR) for which the splash screen is displayed. You'll find this project, `Splash.dpr`, on the accompanying CD-ROM.

LISTING 4.10 A DPR File with a Splash Screen

```
program splash;

uses
  Forms,
  MainFrm in 'MainFrm.pas' {MainForm},
  SplashFrm in 'SplashFrm.pas' {SplashForm};

{$R *.RES}
begin
  Application.Initialize;
  { Create the splash screen }
  SplashForm := TSplashForm.Create(Application);
  SplashForm.Show;   // Display the splash screen
  SplashForm.Update; // Update the splash screen to ensure it gets drawn

  { This while loop simply uses the TTimer component on the SplashForm
    to simulate a lengthy process. }
  while SplashForm.tmMainTimer.Enabled do
    Application.ProcessMessages;

  Application.CreateForm(TMainForm, MainForm);
  SplashForm.Hide;  // Hide the splash screen
  SplashForm.Free;  // Free the splash screen
  Application.Run;
end.
```

Notice the while loop:

```
while SplashForm.tmMainTimer.Enabled do
    Application.ProcessMessages;
```

This is simply a way to simulate a long process. A TTimer component was placed on SplashForm, and its Interval property was set to 3000. When the OnTimer event of the TTimer component occurs, after about three seconds, it executes the following line:

```
tmMainTimer.Enabled := False;
```

This will cause the while loop's condition to be False and will jump execution out of the loop.

Minimizing Form Size

To illustrate how to suppress or control form sizing, we've created a project whose main form has a blue background and a panel onto which components are placed. When the user resizes the form, the panel remains centered. The form also prevents the user from shrinking the form smaller than its panel. Listing 4.11 shows the form's unit source code.

LISTING 4.11 The Source Code for the Template Form

```
unit BlueBackFrm;

interface

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

type
  TBlueBackForm = class(TForm)
    pnlMain: TPanel;
    bbtnOK: TBitBtn;
    bbtnCancel: TBitBtn;
    procedure FormResize(Sender: TObject);
  private
    Procedure CenterPanel;
    { Create a message handler for the WM_WINDOWPOSCHANGING message }
    procedure WMWindowPosChanging(var Msg: TWMWindowPosChanging);
       message WM_WINDOWPOSCHANGING;
  end;

var
  BlueBackForm: TBlueBackForm;

implementation
```

```
uses Math;
{$R *.DFM}

procedure TBlueBackForm.CenterPanel;
{ This procedure centers the main panel horizontally and
  vertically inside the form's client area
}
begin
  { Center horizontally }
  if pnlMain.Width < ClientWidth then
    pnlMain.Left := (ClientWidth - pnlMain.Width) div 2
  else
    pnlMain.Left := 0;

  { Center vertically }
  if pnlMain.Height < ClientHeight then
    pnlMain.Top := (ClientHeight - pnlMain.Height) div 2
  else
    pnlMain.Top := 0;
end;

procedure TBlueBackForm.WMWindowPosChanging(var Msg: TWMWindowPosChanging);
var
  CaptionHeight: integer;
begin
  { Calculate the caption height }
  CaptionHeight := GetSystemMetrics(SM_CYCAPTION);
  { This procedure does not take into account the width and
    height of the form's frame. You can use
    GetSystemMetrics() to obtain these values. }


  // Prevent window from shrinking smaller then MainPanel's width
  Msg.WindowPos^.cx := Max(Msg.WindowPos^.cx, pnlMain.Width+20);

  // Prevent window from shrinking smaller then MainPanel's width
  Msg.WindowPos^.cy := Max(Msg.WindowPos^.cy, pnlMain.Height+20+CaptionHeight);

  inherited;
end;

procedure TBlueBackForm.FormResize(Sender: TObject);
begin
  CenterPanel; // Center MainPanel when the form is resized.
end;

end.
```

This form illustrates capturing window messages, specifically the `WM_WINDOWPOSCHANGING` message, which occurs whenever the window size is about to be changed. This is an opportune time to prevent the resizing of a window. Chapter 5, "Understanding Messages," will delve further into Windows messages. This demo can be found in the project `TempDemo.dpr` on the CD-ROM.

Running a Formless Project

The form is the focal point of all Delphi 5 applications. However, nothing prevents you from creating an application that has no form. The DPR file is nothing more than a program file that "uses" units that define the forms and other objects. This program file can certainly perform other programming processes that require no form. To do this, simply create a new project and remove the main form from the project by selecting Project, Remove From Project. Your DPR file will now contain the following code:

```
program Project1;
uses
 Forms;
{$R *.RES}
begin
  Application.Initialize;
  Application.Run;
end.
```

In fact, you can even remove the `uses` clause and the calls to `Application.Initialize` and `Application.Run`:

```
program Project1;
begin
end.
```

This is a rather useless project, but keep in mind that you can place pretty much whatever you want in the `begin..end` block, which would be the starting point of a Win32 console application.

Exiting Windows

One reason you might want to exit Windows from an application is because your application has made some system configuration changes that don't go into effect until the user restarts Windows. Rather than have the user perform that task through Windows, your application can ask the user whether he or she wants to exit Windows; your application can then take care of the dirty work. Keep in mind, however, that requiring a system restart is considered bad form and should be avoided.

Exiting Windows requires the use of one of two Windows API functions: `ExitWindows()` or `ExitWindowsEx()`.

The ExitWindows() function is a carryover from 16-bit Windows. In that previous version of Windows, you could specify various options that allowed you to reboot Windows after exiting. However, in Win32, this function just logs the current user out of Windows and enables another user to log on to the next Windows session.

ExitWindows() has been replaced by the new function ExitWindowsEx(). With this function, you can log off, shut down Windows, or shut down Windows and restart the system (reboot). Listing 4.12 shows the use of both functions.

LISTING 4.12 Exiting Windows Using ExitWindows() and ExitWindowsEx()

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    btnExit: TButton;
    rgExitOptions: TRadioGroup;
    procedure btnExitClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnExitClick(Sender: TObject);
begin
  case rgExitOptions.ItemIndex of
    0: Win32Check(ExitWindows(0, 0)); // Exit and log on as a
                                      // different user.
    1: Win32Check(ExitWindowsEx(EWX_REBOOT, 0));  // Exit/Reboot
    2: Win32Check(ExitWindowsEx(EWX_SHUTDOWN, 0));// Exit to Power Off
    // Exit/Log off/Log on as different user
    3: Win32Check(ExitWindowsEx(EWX_LOGOFF, 0));
  end;
end;

end.
```

Listing 4.12 uses the value of a radio button to determine which Windows exit option to use. The first option uses `ExitWindows()` to log the user off and restart Windows, asking the user to log on again.

The remaining options use the `ExitWindowsEx()` function. The second option exits Windows and reboots the system. The third option exits Windows and shuts down the system so that the user can turn off the computer. The fourth option performs the same task as the first, except that it uses the `ExitWindowsEx()` function.

Both `ExitWindows()` and `ExitWindowsEx()` return `True` if successful and `False` otherwise. You can use the `Win32Check()` function from `SysUtils.pas`, which calls the Win32 API function `GetLastError()` and displays the proper error string in the event of an error.

NOTE

If you're running Windows NT, the `ExitWindowsEx()` function will not shut down the system; this requires a special privilege. You must use the Win32 API function `AdjustTokenPrivleges()` to enable the `SE_SHUTDOWN_NAME` privilege. More information on this topic can be found in the Win32 online help.

You'll find an example of this code in the project `ExitWin.dpr` on the CD-ROM accompanying this book.

Preventing Windows Shutdown

Shutting down Windows is one thing, but what if another application performs the same task—that is, calls `ExitWindowsEx()`—while you're editing a file and haven't yet saved the file? Unless you somehow capture the exit request, you chance losing valuable data. It's simple to capture the exit request. All that's required is that you process the `OnCloseQuery` event for the main form in your application. In that event handler, you can place code similar to the following:

```
procedure TMainForm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if MessageDlg('Shutdown?', mtConfirmation, mbYesNoCancel, 0) = mrYes then
    CanClose := True
  else
    CanClose := False;
end;
```

By setting `CanClose` to `False`, you tell Windows not to shut down. Another option is to set `CanClose` to `True` only after prompting you to save a file if necessary. You'll find this demonstrated in the project `NoClose.dpr` on the accompanying CD-ROM.

NOTE

If you're running a formless project, you must subclass the application's window procedure and capture the `WM_QUERYENDSESSION` message that's sent to each application running whenever `ExitWindows()` or `ExitWindowsEx()` is called from any application. If the application returns a nonzero value from this message, that application can end successfully. The application should return zero to prevent Windows from shutting down. You'll learn more about processing Windows messages in Chapter 5, "Understanding Messages."

Summary

This chapter focuses on project management techniques and architectural issues. It discusses the key components that make up most Delphi 5 projects: `TForm`, `TApplication`, and `TScreen`. We demonstrated how you might start designing your applications by first developing a common architecture. The chapter also shows various useful routines for your application.