Using the Open Tools API

CHAPTER

17

IN THIS CHAPTER

Have you ever thought to yourself, "Delphi is great, but why doesn't the IDE perform this little task that I'd like it to?" If you have, the Open Tools API is for you. The Delphi Open Tools API provides you with the capability of integrating your own tools that work closely with Delphi's IDE. In this chapter, you'll learn about the different interfaces that make up the Open Tools API, how to use the interfaces, and also how to leverage your newly found expertise to write a fully featured wizard.

Open Tools Interfaces

The Open Tools API is composed of 14 units, each containing one or more objects that provide interfaces to a variety of facilities in the IDE. Using these interfaces enables you to write your own Delphi wizards, version control managers, and component and property editors. You'll also gain a window into Delphi's IDE and editor through any of these add-ons.

With the exception of the interfaces designed for component and property editors, the Open Tools interface objects provide an all-virtual interface to the outside world—meaning that using these interface objects involves working only with the objects' virtual functions. You can't access the objects' data fields, properties, or static functions. Because of this, the Open Tools interface objects follow the COM standard (see Chapter 15, "COM Development"). With a little work on your part, these interfaces can be used in any programming language that supports COM interfaces. In this chapter, you'll work only with Delphi, but you should know that the capacity for using other languages is available (in case you just can't get enough of C++).

NOTE

The complete Open Tools API is available only with Delphi Professional and Enterprise. Delphi Personal has the capability to use add-ons created with the Open Tools API, but it cannot create add-ons because it contains only the units for creating component and property editors. You can find the source code for the Open Tools interfaces in the `\Delphi 6\Source\ToolsAPI` subdirectory.

Table 17.1 shows the units that make up the Open Tools API and the classes and interfaces they provide. Table 17.2 lists obsolete Open Tools API units that remain only for backward compatibility with experts written in Delphi 4 or earlier. Because the obsolete units pre-date the native `interface` type, they employ regular Delphi classes with virtual abstract methods as a substitute for true interfaces. The use of true interfaces has been phased into the Open Tools API over the past few versions of Delphi, and the current incarnation of the Open Tools API is primarily `interface`-based.

TABLE 17.1 Units in the Open Tools API

Unit Name	*Purpose*
`ToolsAPI`	Contains the latest interface-based Open Tools API elements. The contents of this unit essentially supersede the pre-Delphi 5 Open Tools API units that use abstract classes to manipulate menus, notifications, the filesystem, the editor, and wizard add-ins. It also contains new interfaces for manipulating the debugger, IDE key mappings, projects, project groups, packages, and the To Do list.
`VCSIntf`	Defines the `TIVCSClient` class, which enables the Delphi IDE to communicate with version-control software.
`DesignConst`	Contains strings used by the Open Tools API.
`DesignEditors`	Provides property editor support.
`DesignIntf`	This unit replaces the `DsgnIntf` unit from previous versions and provides core support for design-time IDE interfaces. The `IProperty` interface is used by the IDE to edit properties. `IDesignerSelections` is used to manipulate the form designer's selected objects list (replaces `TDesignerSelectionList` used in previous Delphi versions). `IDesigner` is one of the primary interfaces used by wizards for general IDE services. `IDesignNotification` provides notification of designer events such as items being inserted, deleted, or modified. The `IComponentEditor` interface is implemented by component editors to provide design time component editing, and `ISelectionEditor` provides the same functionality for a group of selected components. The `TBaseComponentEditor` class is the class from which all component editors should be derived. `ICustomModule` and `TBaseCustomModule` are provided in order to install modules that can be edited in the IDE's form designer.
`DesignMenus`	Contains the `IMenuItems`, `IMenuItem`, and related interfaces for design-time manipulation of the IDE's menus.
`DesignWindows`	Declares the `TDesignWindow` class, which would serve as the base class for any new design windows one might want to add to the IDE.
`PropertyCategories`	Contains the classes to support the categorization of custom component properties. Used by the Object Inspector's category view.
`TreeIntf`	Provides `TSprig` and related classes and interfaces to support custom *sprigs*, or nodes in the IDE's Object TreeView.
`VCLSprigs`	Sprig implementations for VCL components.

TABLE 17.1 Continued

Unit Name	*Purpose*
`VCLEditors`	Declares base `ICustomPropertyDrawing` and `ICustomPropertyListDrawing` to handle custom drawing of properties and property lists in the IDE's Object Inspector. Also declares custom property drawing objects for common VCL properties.
`ClxDesignWindows`	Declares the `TClxDesignWindow` class, which is the CLX equivalent of the `TDesignWindow` class.
`ClxEditors`	CLX equivalent of the `VCLEditors` unit, which includes property editors for CLX components.
`ClxSprigs`	Sprig implementations for CLX components.

TABLE 17.2 Obsolete Open Tools API units

Unit Name	*Purpose*
`FileIntf`	Defines the `TIVirtualFileSystem` class, which the Delphi IDE uses for filing. Wizards, version-control managers, and property and component editors can use this interface to hook into Delphi's own file system to perform special file operations.
`EditIntf`	Defines classes necessary for manipulating the Delphi Code Editor and Form Designer. The `TIEditReader` class provides read access to an editor buffer. `TIEditWriter` provides write access to the same. `TIEditView` is defined as an individual view of an edit buffer. `TIEditInterface` is the base interface to the editor, which can be used to obtain the previously mentioned editor interfaces. The `TIComponentInterface` class is an interface to an individual component sitting on a form at design time. `TIFormInterface` is the base interface to a design-time form or data module. `TIResourceEntry` is an interface for the raw data in a project's resource (`*.res`) file. `TIResourceFile` is a higher-level interface to the project resource file. `TIModuleNotifier` is a class that provides notifications when various events occur for a particular module. Finally, `TIModuleInterface` is the interface for any file or module open in the IDE.
`ExptIntf`	Defines the abstract `TIExpert` class from which all experts descend.

TABLE 17.2 Continued

Unit Name	*Purpose*
`VirtIntf`	Defines the base `TInterface` class from which other interfaces are derived. This unit also defines `TIStream` class, which is a wrapper around a VCL `TStream`.
`IStreams`	Defines `TIMemoryStream`, `TIFileStream`, and `TIVirtualStream` classes, which are descendants of `TIStream`. These interfaces can be used to hook into the IDE's own streaming mechanism.
`ToolIntf`	Defines `TIMenuItemIntf` and `TIMainMenuIntf` classes, which enable the Open Tools developer to create and modify menus in the Delphi IDE. This unit also defines the `TIAddInNotifier` class, which allows add-in tools to be notified of certain events within the IDE. Most importantly, this unit defines the `TIToolServices` class, which provides an interface into various portions of the Delphi IDE (such as the editor, component library, Code Editor, Form Designer, and filesystem).

NOTE

You might wonder where all this wizard stuff is documented in Delphi. We assure you that it is documented, but the documentation isn't easy to find. Each of these units contains complete documentation for the interface, classes, methods, and procedures declared within. We won't regurgitate the same information that these units contain, so we urge you to take a look at the units for complete documentation.

Using the Open Tools API

Now that you know what's what, it's time to get your hands dirty and look at some actual code. This section focuses primarily on writing wizards by using the Open Tools API. We won't discuss the building of version-control systems because the interest for such a topic is arguably limited. For examples of component and property editors, you should look at Chapter 11, "VCL Component Building," and Chapter 12, "Advanced VCL Component Building."

A Dumb Wizard

To start out, you'll create a very simple wizard appropriately dubbed the Dumb Wizard. The minimum requirement to create a wizard is to create a class that implements the `IOTAWizard` interface. For reference, `IOTAWizard` is defined in the `ToolsAPI` unit as follows:

```
type
  IOTAWizard = interface(IOTANotifier)
```

```
    ['{B75C0CE0-EEA6-11D1-9504-00608CCBF153}']
    { Expert UI strings }
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    { Launch the AddIn }
    procedure Execute;
  end;
```

This interface consists mainly of some `Get`*`XXX`*`()` functions that are designed to be overridden by the descendant classes in order to provide specific information for each wizard. The `Execute()` method is the business end of `IOTAWizard`. `Execute()` is called by the IDE when the user selects your wizard from the main menu or the New Items menu, and it's in this method that the wizard should be created and invoked.

If you've got a keen eye, you might have noticed that `IOTAWizard` descends from another interface, called `IOTANotifier`. `IOTANotifier` is an interface defined in the `ToolsAPI` unit that contains methods that can be called by the IDE to notify a wizard of various goings on. This interface is defined as

```
type
  IOTANotifier = interface(IUnknown)
    ['{F17A7BCF-E07D-11D1-AB0B-00C04FB16FB3}']
    { This procedure is called immediately after the item is successfully
      saved. This is not called for IOTAWizards }
    procedure AfterSave;
    { This function is called immediately before the item is saved. This is not
      called for IOTAWizard }
    procedure BeforeSave;
    { The associated item is being destroyed so all references should be
      dropped. Exceptions are ignored. }
    procedure Destroyed;
    { This associated item was modified in some way. This is not called for
      IOTAWizards }
    procedure Modified;
  end;
```

As the comments in the source code indicate, most of these methods aren't called for simple `IOTAWizard` wizards. Because of this, `ToolsAPI` provides a class called `TNotifierObject` that provides empty implementations for `IOTANotifier` methods. You might choose to descend your wizards from this class to take advantage of the convenience of having the `IOTANotifier` methods implemented for you.

Wizards are not much use without a means to invoke them, and one of the simplest ways to do that is through a menu pick. If you want to place your wizard on Delphi's main menu, you need only implement the `IOTAMenuWizard` interface, which is defined in all its complexity in `ToolsAPI` as

```
type
  IOTAMenuWizard = interface(IOTAWizard)
    ['{B75C0CE2-EEA6-11D1-9504-00608CCBF153}']
    function GetMenuText: string;
  end;
```

As you can see, this interface descends from `IOTAWizard` and adds only one additional method to return the menu text string.

To jump right in and pull together your knowledge thus far, Listing 17.1 shows the `DumbWiz.pas` unit, which contains the source code for `TDumbWizard`.

LISTING 17.1 `DumbWiz.pas`—a Simple Wizard Implementation

```
unit DumbWiz;

interface

uses
  ShareMem, SysUtils, Windows, ToolsAPI;

type
  TDumbWizard = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    // IOTAWizard methods
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // IOTAMenuWizard method
    function GetMenuText: string;
  end;

implementation

uses Dialogs;

function TDumbWizard.GetName: string;
begin
  Result := 'Dumb Wizard';
end;
```

LISTING 17.1 Continued

```
function TDumbWizard.GetState: TWizardState;
begin
  Result := [wsEnabled];
end;

function TDumbWizard.GetIDString: String;
begin
  Result := 'DDG.DumbWizard';
end;

procedure TDumbWizard.Execute;
begin
  MessageDlg('This is a dumb wizard.', mtInformation, [mbOk], 0);
end;

function TDumbWizard.GetMenuText: string;
begin
  Result := 'Dumb Wizard';
end;

end.
```

The `IOTAWizard.GetName()` function should return a unique name for this wizard.

`IOTAWizard.GetState()` returns the state of an `wsStandard` wizard on the main menu. The return value of this function is a set that can contain `wsEnabled` and/or `wsChecked`, depending on how you want the menu item to appear in the IDE. This function is called every time the wizard is shown in order to determine how to paint the menu.

`IOTAWizard.GetIDString()` should return a globally unique string identifier for the wizard. Convention dictates that the return value of this string should be in the following format:

```
CompanyName.WizardName
```

`IOTAWizard.Execute()` invokes the wizard. As Listing 17.1 shows, the `Execute()` method for `TDumbWizard` doesn't do much. However, later in this chapter you'll see some wizards that actually do perform stuff.

`IOTAMenuWizard.GetMenuText()` returns the text that should appear on the main menu. This function is called every time the user pulls down the Help menu, so it's possible to dynamically change the value of the menu text as your wizard runs.

Take a look at the call to `RegisterPackageWizard()` inside the `Register()` procedure. You might notice that this is very similar to the syntax used for registering components, component editors, and property editors for inclusion in the component library, as described in Chapters 11

and 12. The reason for this similarity is that this type of wizard is stored in a package that's part of the component library, along with components and the like. You can also store wizards in a standalone DLL, as you'll see in the next example.

This wizard is installed just like a component: Select the Components, Install Component option from the main menu and add the unit to a new or existing package. Once this is installed, the menu choice to invoke the wizard appears under the Help menu, as shown in Figure 17.1. You can see the outstanding output of this wizard in Figure 17.2.

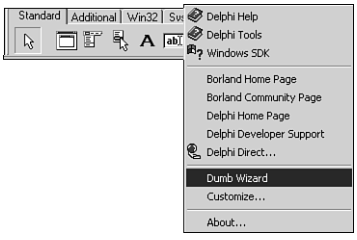

FIGURE 17.1
The Dumb Wizard on the main menu.

FIGURE 17.2
The Dumb Wizard in action.

The Wizard Wizard

There's just a little bit more work involved in creating a DLL-based wizard (as opposed to a component-library–based wizard). In addition to demonstrating the creation of a DLL-based wizard, the Wizard Wizard example has a couple of ulterior motives, including illustrating how DLL wizards relate to the Registry and how to maintain one source code base that targets either an EXE or a DLL wizard.

NOTE

If you're unfamiliar with the ins and outs of Windows DLLs, take a look at Chapter 9, "Dynamic Link Libraries," in the electronic version of *Delphi 5 Developer's Guide* on the CD accompanying this book.

TIP

There's no hard-and-fast rule that dictates whether a wizard should reside in a package in the component library or a DLL. From a user's perspective, the primary difference between the two is that component library wizards require a simple package installation to be rebuilt, whereas DLL wizards require a Registry entry, and Delphi must be exited and restarted for changes to take effect. However, as a developer, you'll find package wizards a bit easier to deal with for a number of reasons. Namely, exceptions propagate between your wizard and the IDE automatically, you don't have to use `sharemem.dll` for memory management, you don't have to do anything special to initialize the DLL's application variable, and pop-up hints and mouse enter/exit messages will work properly.

With this in mind, you should consider using a DLL wizard when you want the wizard to install with a minimum amount of work on the part of the end user.

For Delphi to recognize a DLL wizard, it must have an entry in the system Registry under the following key:

```
HKEY_CURRENT_USER\Software\Borland\Delphi\5.0\Experts
```

Figure 17.3 shows sample entries using the Windows RegEdit application.

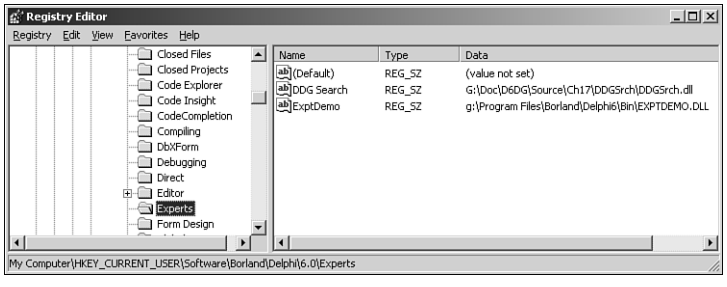

FIGURE 17.3
Delphi wizard entries viewed with RegEdit.

Wizard Interface

The purpose of the Wizard Wizard is to provide an interface to add, modify, and delete DLL wizard entries from the Registry without having to use the cumbersome RegEdit application. First, let's examine `InitWiz.pas`, the unit containing the wizard class (see Listing 17.2).

LISTING 17.2 InitWiz.pas—Unit Containing DLL Wizard Class

```
unit InitWiz;

interface

uses Windows, ToolsAPI;

type
  TWizardWizard = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    // IOTAWizard methods
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // IOTAMenuWizard method
    function GetMenuText: string;
  end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;

var
  { Registry key where Delphi 6 wizards are kept.  EXE version uses default, }
  { whereas DLL version gets key from ToolServices.GetBaseRegistryKey }
  SDelphiKey: string = '\Software\Borland\Delphi\6.0\Experts';

implementation

uses SysUtils, Forms, Controls, Main;
function TWizardWizard.GetName: string;
{ Return name of expert }
begin
  Result := 'WizardWizard';
end;

function TWizardWizard.GetState: TWizardState;
{ This expert is always enabled }
begin
  Result := [wsEnabled];
end;

function TWizardWizard.GetIDString: String;
{ "Vendor.AppName" ID string for expert }
```

LISTING 17.2 Continued

```
begin
  Result := 'DDG.WizardWizard';
end;

function TWizardWizard.GetMenuText: string;
{ Menu text for expert }
begin
  Result := 'Wizard Wizard';
end;

procedure TWizardWizard.Execute;
{ Called when expert is chosen from the main menu. }
{ This procedure creates, shows, and frees the main form. }
begin
  MainForm := TMainForm.Create(Application);
  try
    MainForm.ShowModal;
  finally
    MainForm.Free;
  end;
end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;
var
  Svcs: IOTAServices;
begin
  Result := BorlandIDEServices <> nil;
  if Result then
  begin
    Svcs := BorlandIDEServices as IOTAServices;
    ToolsAPI.BorlandIDEServices := BorlandIDEServices;
    Application.Handle := Svcs.GetParentHandle;
    SDelphiKey := Svcs.GetBaseRegistryKey + '\Experts';
    RegisterProc(TWizardWizard.Create);
  end;
end;

end.
```

You should notice a couple of differences between this unit and the one used to create the Dumb Wizard. Most importantly, an initialization function of type `TWizardInitProc` is required as an entry point for the IDE into the wizard DLL. In this case, that function is called `InitWizard()`. This function performs a number of wizard initialization tasks, including the following:

- Obtaining a `IOTAServices` interface from the `BorlandIDEServices` parameter.
- Saving the `BorlandIDEServices` interface pointer for use at a later time.
- Setting the handle of the DLL's `Application` variable to the value returned by `IOTAServices.GetParentHandle()`. `GetParentHandle()` returns the window handle of the window that must serve as the parent to all top-level windows created by the wizard.
- Passing the newly created instance of the wizard to the `RegisterProc()` procedure in order to register the wizard with the IDE. `RegisterProc()` will be called once for each wizard instance the DLL registers with the IDE.
- Optionally, `InitWizard()` can also assign a procedure of type `TWizardTerminateProc` to the `Terminate` parameter to serve as an exit procedure for the wizard. This procedure will be called immediately before the wizard is unloaded by the IDE, and in it you can perform any necessary cleanup. This parameter is initially `nil`, so if you don't need to perform any special cleanup, leave its value as `nil`.

CAUTION

The wizard initialization method must use the `stdcall` calling convention.

CAUTION

Any DLL wizards calling Open Tools API functions that have string parameters must have the `ShareMem` unit in their `uses` clause; otherwise, Delphi will raise an access violation when the wizard instance is freed.

The Wizard User Interface

The `Execute()` method is a bit more complex this time around. It creates an instance of the wizard's `MainForm`, shows it modally, and then frees the instances. Figure 17.4 shows this form, and Listing 17.3 shows the `Main.pas` unit in which `MainForm` exists.

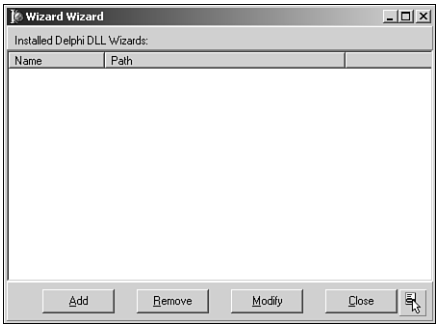

FIGURE 17.4
`MainForm` *in the Wizard Wizard.*

LISTING 17.3 `Main.pas`—Main Unit of Wizard Wizard

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    TopPanel: TPanel;
    Label1: TLabel;
    BottomPanel: TPanel;
    WizList: TListView;
    PopupMenu1: TPopupMenu;
    Add1: TMenuItem;
    Remove1: TMenuItem;
    Modify1: TMenuItem;
    AddBtn: TButton;
    RemoveBtn: TButton;
    ModifyBtn: TButton;
    CloseBtn: TButton;
    procedure RemoveBtnClick(Sender: TObject);
    procedure CloseBtnClick(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure ModifyBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure DoAddMod(Action: TAddModAction);
```

LISTING 17.3 Continued

```
    procedure RefreshReg;
  end;

var
  MainForm: TMainForm;

implementation

uses InitWiz;

{$R *.DFM}

var
  DelReg: TRegistry;

procedure TMainForm.RemoveBtnClick(Sender: TObject);
{ Handler for Remove button click. Removes selected item from registry. }
var
  Item: TListItem;
begin
  Item := WizList.Selected;
  if Item <> nil then
  begin
    if MessageDlg(Format('Remove item "%s"', [Item.Caption]), mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
      DelReg.DeleteValue(Item.Caption);
    RefreshReg;
  end;
end;

procedure TMainForm.CloseBtnClick(Sender: TObject);
{ Handler for Close button click. Closes app. }
begin
  Close;
end;

procedure TMainForm.DoAddMod(Action: TAddModAction);
{ Adds a new expert item to registry or modifies existing one. }
var
  OrigName, ExpName, ExpPath: String;
  Item: TListItem;
begin
  if Action = amaModify then           // if modify...
  begin
    Item := WizList.Selected;
```

LISTING 17.3 Continued

```
    if Item = nil then Exit;           // make sure item is selected
    ExpName := Item.Caption;           // init variables
    if Item.SubItems.Count > 0 then
      ExpPath := Item.SubItems[0];
    OrigName := ExpName;                 // save original name
  end;
  { Invoke dialog which allows user to add or modify entry }
  if AddModWiz(Action, ExpName, ExpPath) then
  begin
    { if action is Modify, and the name was changed, handle it }
    if (Action = amaModify) and (OrigName <> ExpName) then
      DelReg.RenameValue(OrigName, ExpName);
    DelReg.WriteString(ExpName, ExpPath);  // write new value
  end;
  RefreshReg;                              // update listbox
end;

procedure TMainForm.AddBtnClick(Sender: TObject);
{ Handler for Add button click }
begin
  DoAddMod(amaAdd);
end;

procedure TMainForm.ModifyBtnClick(Sender: TObject);
{ Handler for Modify button click }
begin
  DoAddMod(amaModify);
end;

procedure TMainForm.RefreshReg;
{ Refreshes listbox with contents of registry }
var
  i: integer;
  TempList: TStringList;
  Item: TListItem;
begin
  WizList.Items.Clear;
  TempList := TStringList.Create;
  try
    { Get expert names from registry }
    DelReg.GetValueNames(TempList);
    { Get path strings for each expert name }
    for i := 0 to TempList.Count - 1 do
    begin
      Item := WizList.Items.Add;
```

LISTING 17.3 Continued

```
      Item.Caption := TempList[i];
      Item.SubItems.Add(DelReg.ReadString(TempList[i]));
    end;
  finally
    TempList.Free;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  RefreshReg;
end;

initialization
  DelReg := TRegistry.Create;          // create registry object
  DelReg.RootKey := HKEY_CURRENT_USER; // set root key
  DelReg.OpenKey(SDelphiKey, True);    // open/create Delphi expert key
finalization
  Delreg.Free;                         // free registry object
end.
```


This is the unit responsible for providing the user interface for adding, removing, and modifying DLL wizard entries in the Registry. In the `initialization` section of this unit, a `TRegistry` object called `DelReg` is created. The `RootKey` property of `DelReg` is set to `HKEY_CURRENT_USER`, and it opens the `\Software\Borland\Delphi\6.0\Experts` key—the key used to keep track of DLL wizards—using its `OpenKey()` method.

When the wizard first comes up, a `TListView` component called `ExptList` is filled with the items and values from the previously mentioned Registry key. This is accomplished by first calling `DelReg.GetValueNames()` to retrieve the names of the items into a `TStringList`. A `TListItem` component is added to `ExptList` for each element in the string list, and the `DelReg.ReadString()` method is used to read the value for each item, which is placed in the `SubItems` list of `TListItem`.

The Registry work is done in the `RemoveBtnClick()` and `DoAddMod()` methods. `RemoveBtnClick()` is in charge of removing the currently selected wizard item from the Registry. It first checks to ensure that an item is highlighted; then it throws up a confirmation dialog box. Finally, it does the deed by calling the `DelReg.DeleteValue()` method and passing `CurrentItem` as the parameter.

`DoAddMod()` accepts a parameter of type `TAddModAction`. This type is defined as follows:

```
type
  TAddModAction = (amaAdd, amaModify);
```

As the values of the type imply, this variable indicates whether a new item is to be added or an existing item modified. This function first checks to see that there's a currently selected item or, if there isn't, that the `Action` parameter holds the value `amaAdd`. After that, if `Action` is `amaModify`, the existing wizard item and value are copied to the local variables `ExpName` and `ExpPath`. These values are then passed to a function called `AddModExpert()`, which is defined in the `AddModU` unit shown in Listing 17.4. This function invokes a dialog box in which the user can enter new or modified name or path information for a wizard (see Figure 17.5). It returns `True` when the user exits the dialog with the OK button. At that point, an existing item is modified using `DelReg.RenameValue()`, and a new or modified value is written with `DelReg.WriteString()`.

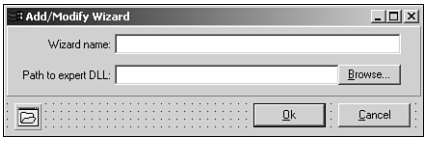

FIGURE 17.5
`AddModForm` *in the Wizard Wizard.*

LISTING 17.4 `AddModU.pas`—Unit That Adds and Modifies Wizard Entries in the Registry

```
unit AddModU;

interface

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

type
  TAddModAction = (amaAdd, amaModify);

  TAddModForm = class(TForm)
    OkBtn: TButton;
    CancelBtn: TButton;
    OpenDialog: TOpenDialog;
    Panel1: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    PathEd: TEdit;
    NameEd: TEdit;
    BrowseBtn: TButton;
    procedure BrowseBtnClick(Sender: TObject);
  private
```

LISTING 17.4 Continued

```
    { Private declarations }
  public
    { Public declarations }
  end;

function AddModWiz(AAction: TAddModAction; var WizName,
  WizPath: String): Boolean;

implementation

{$R *.DFM}

function AddModWiz(AAction: TAddModAction; var WizName,
  WizPath: String): Boolean;
{ called to invoke dialog to add and modify registry entries }
const
  CaptionArray: array[TAddModAction] of string[31] =
    ('Add new expert', 'Modify expert');
begin
  with TAddModForm.Create(Application) do       // create dialog
  begin
    Caption := CaptionArray[AAction];            // set caption
    if AAction = amaModify then                  // if modify...
    begin
      NameEd.Text := WizName;                   // init name and
      PathEd.Text := WizPath;                   // path
    end;
    Result := ShowModal = mrOk;                 // show dialog
    if Result then                              // if Ok...
    begin
      WizName := NameEd.Text;                   // set name and
      WizPath := PathEd.Text;                   // path
    end;
    Free;
  end;
end;

procedure TAddModForm.BrowseBtnClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    PathEd.Text := OpenDialog.FileName;
end;


end.
```

Dual Targets: EXE and DLL

As mentioned earlier, it's possible to maintain one set of source code modules that target both a DLL wizard and a standalone executable. This is possible through the use of compiler directives in the project file. Listing 17.5 shows `WizWiz.dpr`, the project file source code for this project.

LISTING 17.5 `WizWiz.dpr`—Main Project File for the WizWiz Project

```
{$ifdef BUILD_EXE}
program WizWiz;     // Build as EXE
{$else}
library WizWiz;     // Build as DLL
{$endif}

uses
{$ifndef BUILD_EXE}
  ShareMem,                  // ShareMem required for DLL
  InitWiz in 'InitWiz.pas',  // Wizard stuff
{$endif}
  ToolsAPI,
  Forms,
  Main in 'Main.pas' {MainForm},
  AddModU in 'AddModU.pas' {AddModForm};

{$ifdef BUILD_EXE}
{$R *.RES}                              // required for EXE
{$else}
exports                                 // required for DLL
  InitWizard name WizardEntryPoint;     // required entry point
{$endif}

begin
{$ifdef BUILD_EXE}                      // required for EXE...
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
{$endif}
end.
```

As the code shows, this project will build an executable if the `BUILD_EXE` conditional is defined. Otherwise, it will build a DLL-based wizard. You can define a conditional under Conditional Defines in the Directories/Conditionals page of the Project Options dialog box, which is shown in Figure 17.6.

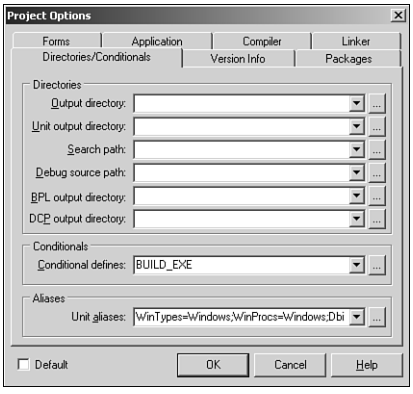

FIGURE 17.6
The Project Options dialog box.

One final note concerning this project: Notice that the `InitWizard()` function from the `InitWiz` unit is being exported in the `exports` clause of the project file. You must export this function with the name `WizardEntryPoint`, which is defined in the `ToolsAPI` unit.

CAUTION

Borland doesn't provide a `ToolsAPI.dcu` file, meaning that EXEs or DLLs containing a reference to `ToolsAPI` in a `uses` clause can only be built *with packages*. It isn't currently possible to build wizards without packages.

DDG Search

Remember the nifty little Delphi Search program you developed back in Chapter 5, " Multithreaded Techniques"? In this section, you'll learn how you can turn that useful application into an even more useful Delphi wizard with just a little bit of code. This wizard is called DDG Search.

First, the unit that interfaces DDG Search to the IDE, `InitWiz.pas`, is shown in Listing 17.6. You'll notice that this unit is very similar to the unit of the same name in the previous example. That's on purpose. This unit is just a copy of the previous one with some necessary changes involving the name of the wizard and the `Execute()` method. Copying and pasting is what we call "old-fashioned inheritance." After all, why do more typing than you have to?

LISTING 17.6 `InitWiz.pas`—Unit Containing Wizard Logic for the `DDGSrch` Wizard

```
unit InitWiz;

interface
```

LISTING 17.6 Continued

```
uses
  Windows, ToolsAPI;

type
  TSearchWizard = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    // IOTAWizard methods
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // IOTAMenuWizard method
    function GetMenuText: string;
  end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;

var
  ActionSvc: IOTAActionServices;

implementation

uses SysUtils, Dialogs, Forms, Controls, Main, PriU;

function TSearchWizard.GetName: string;
{ Return name of expert }
begin
  Result := 'DDG Search';
end;

function TSearchWizard.GetState: TWizardState;
{ This expert is always enabled on the menu }
begin
  Result := [wsEnabled];
end;

function TSearchWizard.GetIDString: String;
{ Return the unique Vendor.Product name of expert }
begin
  Result := 'DDG.DDGSearch';
end;

function TSearchWizard.GetMenuText: string;
{ Return text for Help menu }
```

LISTING 17.6 Continued

```
begin
  Result := 'DDG Search Expert';
end;

procedure TSearchWizard.Execute;
{ Called when expert name is selected from Help menu of IDE. }
{ This function invokes the expert }
begin
  // if not created, created it and show it
  if MainForm = nil then
  begin
    MainForm := TMainForm.Create(Application);
    ThreadPriWin := TThreadPriWin.Create(Application);
    MainForm.Show;
  end
  else
  // if created then restore window and show it
    with MainForm do
    begin
      if not Visible then Show;
      if WindowState = wsMinimized then WindowState := wsNormal;
      SetFocus;
    end;
end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;
var
  Svcs: IOTAServices;
begin
  Result := BorlandIDEServices <> nil;
  if Result then
  begin
    Svcs := BorlandIDEServices as IOTAServices;
    ActionSvc := BorlandIDEServices as IOTAActionServices;
    ToolsAPI.BorlandIDEServices := BorlandIDEServices;
    Application.Handle := Svcs.GetParentHandle;
    RegisterProc(TSearchWizard.Create);
  end;
end;

end.
```

The `Execute()` function of this wizard shows you something a bit different from what you've seen so far: The wizard's main form, `MainForm`, is being shown modelessly rather than modally. Of course, this requires a bit of extra housekeeping because you have to know when a form is created and when the form variable is invalid. This can be accomplished by making sure that the `MainForm` variable is set to `nil` when the wizard is inactive. More on this is discussed a bit later.

One other aspect of this project that has changed significantly since Chapter 5 is that the project file is now called `DDGSrch.dpr`. This file is shown in Listing 17.7.

LISTING 17.7 `DDGSrch.dpr`—Project File for the `DDGSrch` Project

```
{$IFDEF BUILD_EXE}
program DDGSrch;
{$ELSE}
library DDGSrch;
{$ENDIF}

uses
{$IFDEF BUILD_EXE}
  Forms,
{$ELSE}
  ShareMem,
  ToolsAPI,
  InitWiz in 'InitWiz.pas',
{$ENDIF}
  Main in 'MAIN.PAS' {MainForm},
  SrchIni in 'SrchIni.pas',
  SrchU in 'SrchU.pas',
  PriU in 'PriU.pas' {ThreadPriWin},
  MemMap in '..\..\Utils\MemMap.pas',
  DDGStrUtils in '..\..\Utils\DDGStrUtils.pas';

{$R *.RES}

{$IFNDEF BUILD_EXE}
exports
  { Entry point which is called by Delphi IDE }
  InitWizard name WizardEntryPoint;
{$ENDIF}

begin
{$IFDEF BUILD_EXE}
  Application.Initialize;
```

LISTING 17.7 Continued

```
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
{$ENDIF}
end.
```

Once again, you can see that this project is designed to be compiled as a standalone EXE or a DLL-based wizard. When compiled as a wizard, it uses the `library` header to indicate that it's a DLL, and it exports the `InitWiz()` function for initialization by the Delphi IDE.

We made only a couple of changes to the `Main` unit in this project. As mentioned earlier, the `MainForm` variable must be set to `nil` when the wizard isn't active. As you learned in Chapter 2, "The Object Pascal Language," the `MainForm` instance variable will automatically have the value `nil` upon application startup. Also, in the `OnClose` event handler for the form, the form instance is released and the `MainForm` global is reset to `nil`. Here's the method:

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  Application.OnShowHint := FOldShowHint;
  MainForm := nil;
end;
```

The finishing touch for this wizard is to bring up files in the IDE's Code Editor when they're double-clicked in the list box in the main form. This logic is handled by a new `FileLBDblClick()` method, as follows:

```
procedure TMainForm.FileLBDblClick(Sender: TObject);
{ Called when user double-clicks in listbox. Loads file into IDE }
var
  FileName: string;
  Len: Integer;
begin
  { make sure user clicked on a file... }
  if Integer(FileLB.Items.Objects[FileLB.ItemIndex]) > 0 then
  begin
    FileName := FileLB.Items[FileLB.ItemIndex];
    { Trim "File " and ":" from string }
    FileName := Copy(FileName, 6, Length(FileName));
    Len := Length(FileName);
    if FileName[Len] = ':' then SetLength(FileName, Len - 1);
    { Open the project or file }
{$IFNDEF BUILD_EXE}
    if CompareText(ExtractFileExt(FileName), '.DPR') = 0 then
      ActionSvc.OpenProject(FileName, True)
```

```
    else
      ActionSvc.OpenFile(FileName);
{$ELSE}
    ShellExecute(0, 'open', PChar(FileName), nil, nil, SW_SHOWNORMAL);
{$ENDIF}
  end;
end;
```

When compiled as a wizard, this method employs the `OpenFile()` and `OpenProject()` methods of the `IOTAActionServices` in order to open a particular file. As a standalone EXE, this method calls the `ShellExecute()` API function to open the file using the default application associated with the file extension.

Listing 17.8 shows the complete source code for the `Main` unit in the `DDGSrch` project, and Figure 17.7 shows the DDG Search Wizard doing its thing inside the IDE.

LISTING 17.8 `Main.pas`—the Main Unit for the `DDGSrch` Project

```
unit Main;

interface

{$WARN UNIT_PLATFORM OFF}

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, ExtCtrls, Menus, SrchIni,
  SrchU, ComCtrls;

type
  TMainForm = class(TForm)
    FileLB: TListBox;
    PopupMenu1: TPopupMenu;
    Font1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    FontDialog1: TFontDialog;
    StatusBar: TStatusBar;
    AlignPanel: TPanel;
    ControlPanel: TPanel;
    ParamsGB: TGroupBox;
    LFileSpec: TLabel;
    LToken: TLabel;
    lPathName: TLabel;
    EFileSpec: TEdit;
    EToken: TEdit;
```

LISTING 17.8 Continued

```
    PathButton: TButton;
    OptionsGB: TGroupBox;
    cbCaseSensitive: TCheckBox;
    cbFileNamesOnly: TCheckBox;
    cbRecurse: TCheckBox;
    SearchButton: TBitBtn;
    CloseButton: TBitBtn;
    PrintButton: TBitBtn;
    PriorityButton: TBitBtn;
    View1: TMenuItem;
    EPathName: TEdit;
    procedure SearchButtonClick(Sender: TObject);
    procedure PathButtonClick(Sender: TObject);
    procedure FileLBDrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure Font1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure PrintButtonClick(Sender: TObject);
    procedure CloseButtonClick(Sender: TObject);
    procedure FileLBDblClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure PriorityButtonClick(Sender: TObject);
    procedure ETokenChange(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    FOldShowHint: TShowHintEvent;
    procedure ReadIni;
    procedure WriteIni;
    procedure DoShowHint(var HintStr: string; var CanShow: Boolean;
      var HintInfo: THintInfo);
  protected
    procedure WndProc(var Message: TMessage); override;
  public
    Running: Boolean;
    SearchPri: integer;
    SearchThread: TSearchThread;
    procedure EnableSearchControls(Enable: Boolean);
  end;

var
  MainForm: TMainForm;

implementation
```

LISTING 17.8 Continued

```
{$R *.DFM}

uses Printers, ShellAPI, MemMap, FileCtrl, PriU;

procedure PrintStrings(Strings: TStrings);
{ This procedure prints all of the string in the Strings parameter }
var
  Prn: TextFile;
  i: word;
begin
  if Strings.Count = 0 then // Are there strings?
  begin
    MessageDlg('No text to print!', mtInformation, [mbOk], 0);
    Exit;
  end;
  AssignPrn(Prn);                          // assign Prn to printer
  try
    Rewrite(Prn);                          // open printer
    try
      for i := 0 to Strings.Count - 1 do   // iterate over all strings
        WriteLn(Prn, Strings.Strings[i]);  // write to printer
    finally
      CloseFile(Prn);                      // close printer
    end;
  except
    on EInOutError do
      MessageDlg('Error Printing text.', mtError, [mbOk], 0);
  end;
end;

procedure TMainForm.EnableSearchControls(Enable: Boolean);
{ Enables or disables certain controls so options can't be modified }
{ while search is executing. }
begin
  SearchButton.Enabled := Enable;        // enabled/disable proper controls
  cbRecurse.Enabled := Enable;
  cbFileNamesOnly.Enabled := Enable;
  cbCaseSensitive.Enabled := Enable;
  PathButton.Enabled := Enable;
  EPathName.Enabled := Enable;
  EFileSpec.Enabled := Enable;
  EToken.Enabled := Enable;
  Running := not Enable;                 // set Running flag
  ETokenChange(nil);
```

LISTING 17.8 Continued

```
  with CloseButton do
  begin
    if Enable then
    begin                 // set props of Close/Stop button
      Caption := '&Close';
      Hint := 'Close Application';
    end
    else begin
      Caption := '&Stop';
      Hint := 'Stop Searching';
    end;
  end;
end;

procedure TMainForm.SearchButtonClick(Sender: TObject);
{ Called when Search button is clicked.  Invokes search thread. }
begin
  EnableSearchControls(False);          // disable controls
  FileLB.Clear;                         // clear listbox
  { start thread }
  SearchThread := TSearchThread.Create(cbCaseSensitive.Checked,
    cbFileNamesOnly.Checked, cbRecurse.Checked, EToken.Text,
    EPathName.Text, EFileSpec.Text, Handle);
end;

procedure TMainForm.ETokenChange(Sender: TObject);
begin
  SearchButton.Enabled := not Running and (EToken.Text <> '');
end;

procedure TMainForm.PathButtonClick(Sender: TObject);
{ Called when Path button is clicked.  Allows user to choose new path. }
var
  ShowDir: string;
begin
  ShowDir := EPathName.Text;
  if SelectDirectory(ShowDir, [], 0) then
    EPathName.Text := ShowDir;
end;

procedure TMainForm.FileLBDblClick(Sender: TObject);
{ Called when user double-clicks in listbox. Loads file into IDE }
var
  FileName: string;
  Len: Integer;
```

LISTING 17.8 Continued

```
begin
  { make sure user clicked on a file... }
  if Integer(FileLB.Items.Objects[FileLB.ItemIndex]) > 0 then
  begin
    FileName := FileLB.Items[FileLB.ItemIndex];
    { Trim "File " and ":" from string }
    FileName := Copy(FileName, 6, Length(FileName));
    Len := Length(FileName);
    if FileName[Len] = ':' then SetLength(FileName, Len - 1);
    { Open the project or file }
{$IFNDEF BUILD_EXE}
    if CompareText(ExtractFileExt(FileName), '.DPR') = 0 then
      ActionSvc.OpenProject(FileName, True)
    else
      ActionSvc.OpenFile(FileName);
{$ELSE}
    ShellExecute(0, 'open', PChar(FileName), nil, nil, SW_SHOWNORMAL);
{$ENDIF}
  end;
end;

procedure TMainForm.FileLBDrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
{ Called in order to owner draw listbox. }
var
  CurStr: string;
begin
  with FileLB do
  begin
    CurStr := Items.Strings[Index];
    Canvas.FillRect(Rect);                  // clear out rect
    if not cbFileNamesOnly.Checked then     // if not filename only...
    begin
      { if current line is file name... }
      if Integer(Items.Objects[Index]) > 0 then
        Canvas.Font.Style := [fsBold]; // bold font
      end
    else
      Rect.Left := Rect.Left + 15;          // otherwise, indent
    DrawText(Canvas.Handle, PChar(CurStr), Length(CurStr), Rect,
dt_SingleLine);
  end;
end;
```

LISTING 17.8 Continued

```
procedure TMainForm.Font1Click(Sender: TObject);
{ Allows user to pick new font for listbox }
begin
  { Pick new listbox font }
  if FontDialog1.Execute then
    FileLB.Font := FontDialog1.Font;
end;


procedure TMainForm.FormDestroy(Sender: TObject);
{ OnDestroy event handler for form }
begin
  WriteIni;
end;

procedure TMainForm.FormCreate(Sender: TObject);
{ OnCreate event handler for form }
begin
  Application.HintPause := 0;             // don't wait to show hints
  FOldShowHint := Application.OnShowHint; // set up hints
  Application.OnShowHint := DoShowHint;
  ReadIni;                                // read reg INI file
end;

procedure TMainForm.DoShowHint(var HintStr: string; var CanShow: Boolean;
  var HintInfo: THintInfo);
{ OnHint event handler for Application }
begin
  { Display application hints on status bar }
  StatusBar.Panels[0].Text := HintStr;
  { Don't show tool tip if we're over our own controls }
  if (HintInfo.HintControl <> nil) and
    (HintInfo.HintControl.Parent <> nil) and
    ((HintInfo.HintControl.Parent = ParamsGB) or
    (HintInfo.HintControl.Parent = OptionsGB) or
    (HintInfo.HintControl.Parent = ControlPanel)) then
    CanShow := False;
  if Assigned(FOldShowHint) then
    FOldShowHint(HintStr, CanSHow, HintInfo);
end;

procedure TMainForm.PrintButtonClick(Sender: TObject);
{ Called when Print button is clicked. }
begin
  if MessageDlg('Send search results to printer?', mtConfirmation,
```

LISTING 17.8 Continued

```
    [mbYes, mbNo], 0) = mrYes then
    PrintStrings(FileLB.Items);
end;

procedure TMainForm.CloseButtonClick(Sender: TObject);
{ Called to stop thread or close application }
begin
  // if thread is running then terminate thread
  if Running then SearchThread.Terminate
  // otherwise close app
  else Close;
end;


procedure TMainForm.FormResize(Sender: TObject);
{ OnResize event handler. Centers controls in form. }
begin
 { divide status bar into two panels with a 1/3 - 2/3 split }
  with StatusBar do
  begin
    Panels[0].Width := Width div 3;
    Panels[1].Width := Width * 2 div 3;
  end;
  { center controls in the middle of the form }
  ControlPanel.Left := (AlignPanel.Width div 2) - (ControlPanel.Width div 2);
end;

procedure TMainForm.PriorityButtonClick(Sender: TObject);
{ Show thread priority form }
begin
  ThreadPriWin.Show;
end;

procedure TMainForm.ReadIni;
{ Reads default values from Registry }
begin
  with SrchIniFile do
  begin
    EPathName.Text := ReadString('Defaults', 'LastPath', 'C:\');
    EFileSpec.Text := ReadString('Defaults', 'LastFileSpec', '*.*');
    EToken.Text := ReadString('Defaults', 'LastToken', '');
    cbFileNamesOnly.Checked := ReadBool('Defaults', 'FNamesOnly', False);
    cbCaseSensitive.Checked := ReadBool('Defaults', 'CaseSens', False);
    cbRecurse.Checked := ReadBool('Defaults', 'Recurse', False);
```

LISTING 17.8 Continued

```
    Left := ReadInteger('Position', 'Left', 100);
    Top := ReadInteger('Position', 'Top', 50);
    Width := ReadInteger('Position', 'Width', 510);
    Height := ReadInteger('Position', 'Height', 370);
  end;
end;

procedure TMainForm.WriteIni;
{ writes current settings back to Registry }
begin
  with SrchIniFile do
  begin
    WriteString('Defaults', 'LastPath', EPathName.Text);
    WriteString('Defaults', 'LastFileSpec', EFileSpec.Text);
    WriteString('Defaults', 'LastToken', EToken.Text);
    WriteBool('Defaults', 'CaseSens', cbCaseSensitive.Checked);
    WriteBool('Defaults', 'FNamesOnly', cbFileNamesOnly.Checked);
    WriteBool('Defaults', 'Recurse', cbRecurse.Checked);
    WriteInteger('Position', 'Left', Left);
    WriteInteger('Position', 'Top', Top);
    WriteInteger('Position', 'Width', Width);
    WriteInteger('Position', 'Height', Height);
  end;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  Application.OnShowHint := FOldShowHint;
  MainForm := nil;
end;

procedure TMainForm.WndProc(var Message: TMessage);
begin
  if Message.Msg = DDGM_ADDSTR then
  begin
    FileLB.Items.AddObject(PChar(Message.WParam), TObject(Message.LParam));
    StrDispose(PChar(Message.WParam));
  end
  else
    inherited WndProc(Message);
end;

end.
```

TIP

Note the following line from Listing 17.8:

```
{$WARN UNIT_PLATFORM OFF}
```

This compiler directive is used to silence the compile-time warning that is generated because `Main.pas` uses the `FileCtrl` unit, which is a Windows platform specific unit. `FileCtrl` is marked as such using the `platform` directive.

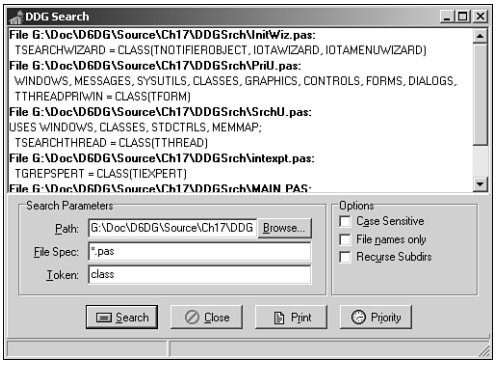

FIGURE 17.7
The DDG Search wizard in action.

Form Wizards

Yet another type of wizard supported by the Open Tools API is the form wizard. Once installed, form wizards are accessed from the New Items dialog box; they generate new forms and units for the user. Chapter 16, "Windows Shell Programming," employs this type of wizard to generate new `AppBar` forms; however, you didn't get to see the code that made the wizard tick.

Creating a form wizard is fairly straightforward, although there a good number of interface methods that you must implement. Creation of a form wizard can be boiled down to five basic steps:

1. Create a class that descends from `TCustomForm`, `TDataModule`, or any `TWinControl` that will be used as the base form class. This class will typically reside in a separate unit from the wizard. In this case, `TAppBar` will serve as the base class.
2. Create a `TNotifierObject` descendent that implements the following interfaces: `IOTAWizard`, `IOTARepositoryWizard`, `IOTAFormWizard`, `IOTACreator`, and `IOTAModuleCreator`.

3. In your `IOTAWizard.Execute()` method, you will typically call `IOTAModuleServices.GetNewModuleAndClassName()` to obtain a new unit and classname for your wizard and `IOTAModuleServices.CreateModule()` to instruct the IDE to begin creation of the new module.
4. Many of the method implementations for the aforementioned interfaces are one-liners. The non-trivial ones include `IOTAModuleCreator`'s `NewFormFile()` and `NewImplFile()` methods, which will return the code for the form and unit, respectively. The `IOTACreator.GetOwner()` method can also be a little tricky, but the example that follows gives you a good technique for adding the unit to the current project (if any).
5. Complete the `Register()` procedure for the wizard by registering a handler for your new form class using the `RegisterCustomModule()` procedure in the `DsgnIntf` unit and creating your wizard by calling the `RegisterPackageWizard()` procedure in the `ToolsAPI` unit.

Listing 17.9 shows the source code for `ABWizard.pas`, which is the `AppBar` wizard.

LISTING 17.9 `ABWizard.pas`—The Unit Containing the Implementation of the `AppBar` Wizard

```
unit ABWizard;

interface

uses Windows, Classes, ToolsAPI;

type
  TAppBarWizard = class(TNotifierObject, IOTAWizard, IOTARepositoryWizard,
    IOTAFormWizard, IOTACreator, IOTAModuleCreator)
  private
    FUnitIdent: string;
    FClassName: string;
    FFileName: string;
  protected
    // IOTAWizard methods
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // IOTARepositoryWizard / IOTAFormWizard methods
    function GetAuthor: string;
    function GetComment: string;
    function GetPage: string;
    function GetGlyph: HICON;
```

LISTING 17.9 Continued

```
    // IOTACreator methods
    function GetCreatorType: string;
    function GetExisting: Boolean;
    function GetFileSystem: string;
    function GetOwner: IOTAModule;
    function GetUnnamed: Boolean;
    // IOTAModuleCreator methods
    function GetAncestorName: string;
    function GetImplFileName: string;
    function GetIntfFileName: string;
    function GetFormName: string;
    function GetMainForm: Boolean;
    function GetShowForm: Boolean;
    function GetShowSource: Boolean;
    function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
    function NewImplSource(const ModuleIdent, FormIdent,
      AncestorIdent: string): IOTAFile;
    function NewIntfSource(const ModuleIdent, FormIdent,
      AncestorIdent: string): IOTAFile;
    procedure FormCreated(const FormEditor: IOTAFormEditor);
  end;

implementation

uses Forms, AppBars, SysUtils, DsgnIntf;

{$R CodeGen.res}

type
  TBaseFile = class(TInterfacedObject)
  private
    FModuleName: string;
    FFormName: string;
    FAncestorName: string;
  public
    constructor Create(const ModuleName, FormName, AncestorName: string);
  end;

  TUnitFile = class(TBaseFile, IOTAFile)
  protected
    function GetSource: string;
    function GetAge: TDateTime;
  end;
```

LISTING 17.9 Continued

```
  TFormFile = class(TBaseFile, IOTAFile)
  protected
    function GetSource: string;
    function GetAge: TDateTime;
  end;

{ TBaseFile }

constructor TBaseFile.Create(const ModuleName, FormName,
  AncestorName: string);
begin
  inherited Create;
  FModuleName := ModuleName;
  FFormName := FormName;
  FAncestorName := AncestorName;
end;

{ TUnitFile }

function TUnitFile.GetSource: string;
var
  Text: string;
  ResInstance: THandle;
  HRes: HRSRC;
begin
  ResInstance := FindResourceHInstance(HInstance);
  HRes := FindResource(ResInstance, 'CODEGEN', RT_RCDATA);
  Text := PChar(LockResource(LoadResource(ResInstance, HRes)));
  SetLength(Text, SizeOfResource(ResInstance, HRes));
  Result := Format(Text, [FModuleName, FFormName, FAncestorName]);
end;

function TUnitFile.GetAge: TDateTime;
begin
  Result := -1;
end;

{ TFormFile }

function TFormFile.GetSource: string;
const
  FormText =
    'object %0:s: T%0:s'#13#10'end';
```

LISTING 17.9 Continued

```
begin
  Result := Format(FormText, [FFormName]);
end;

function TFormFile.GetAge: TDateTime;
begin
  Result := -1;
end;

{ TAppBarWizard }

{ TAppBarWizard.IOTAWizard }

function TAppBarWizard.GetIDString: string;
begin
  Result := 'DDG.AppBarWizard';
end;

function TAppBarWizard.GetName: string;
begin
  Result := 'DDG AppBar Wizard';
end;

function TAppBarWizard.GetState: TWizardState;
begin
  Result := [wsEnabled];
end;

procedure TAppBarWizard.Execute;
begin
  (BorlandIDEServices as IOTAModuleServices).GetNewModuleAndClassName(
    'AppBar', FUnitIdent, FClassName, FFileName);
  (BorlandIDEServices as IOTAModuleServices).CreateModule(Self);
end;

{ TAppBarWizard.IOTARepositoryWizard / TAppBarWizard.IOTAFormWizard }

function TAppBarWizard.GetGlyph: HICON;
begin
  Result := 0;  // use standard icon
end;

function TAppBarWizard.GetPage: string;
```

LISTING 17.9 Continued

```
begin
  Result := 'DDG';
end;

function TAppBarWizard.GetAuthor: string;
begin
  Result := 'Delphi 5 Developer''s Guide';
end;

function TAppBarWizard.GetComment: string;
begin
  Result := 'Creates a new AppBar form.'
end;

{ TAppBarWizard.IOTACreator }

function TAppBarWizard.GetCreatorType: string;
begin
  Result := '';
end;

function TAppBarWizard.GetExisting: Boolean;
begin
  Result := False;
end;

function TAppBarWizard.GetFileSystem: string;
begin
  Result := '';
end;

function TAppBarWizard.GetOwner: IOTAModule;
var
  I: Integer;
  ModServ: IOTAModuleServices;
  Module: IOTAModule;
  ProjGrp: IOTAProjectGroup;
begin
  Result := nil;
  ModServ := BorlandIDEServices as IOTAModuleServices;
  for I := 0 to ModServ.ModuleCount - 1 do
  begin
    Module := ModSErv.Modules[I];
    // find current project group
```

LISTING 17.9 Continued

```
    if CompareText(ExtractFileExt(Module.FileName), '.bpg') = 0 then
      if Module.QueryInterface(IOTAProjectGroup, ProjGrp) = S_OK then
      begin
        // return active project of group
        Result := ProjGrp.GetActiveProject;
        Exit;
      end;
  end;
end;

function TAppBarWizard.GetUnnamed: Boolean;
begin
  Result := True;
end;

{ TAppBarWizard.IOTAModuleCreator }

function TAppBarWizard.GetAncestorName: string;
begin
  Result := 'TAppBar';
end;

function TAppBarWizard.GetImplFileName: string;
var
  CurrDir: array[0..MAX_PATH] of char;
begin
  // Note: full path name required!
  GetCurrentDirectory(SizeOf(CurrDir), CurrDir);
  Result := Format('%s\%s.pas', [CurrDir, FUnitIdent, '.pas']);
end;

function TAppBarWizard.GetIntfFileName: string;
begin
  Result := '';
end;

function TAppBarWizard.GetFormName: string;
begin
  Result := FClassName;
end;

function TAppBarWizard.GetMainForm: Boolean;
begin
  Result := False;
end;
```

LISTING 17.9 Continued

```
function TAppBarWizard.GetShowForm: Boolean;
begin
  Result := True;
end;

function TAppBarWizard.GetShowSource: Boolean;
begin
  Result := True;
end;

function TAppBarWizard.NewFormFile(const FormIdent,
  AncestorIdent: string): IOTAFile;
begin
  Result := TFormFile.Create('', FormIdent, AncestorIdent);
end;

function TAppBarWizard.NewImplSource(const ModuleIdent, FormIdent,
  AncestorIdent: string): IOTAFile;
begin
  Result := TUnitFile.Create(ModuleIdent, FormIdent, AncestorIdent);
end;

function TAppBarWizard.NewIntfSource(const ModuleIdent, FormIdent,
  AncestorIdent: string): IOTAFile;
begin
  Result := nil;
end;

procedure TAppBarWizard.FormCreated(const FormEditor: IOTAFormEditor);
begin
  // do nothing
end;

end.
```


This unit employs an interesting trick for source code generation: The unformatted source code is stored in an `RES` file that's linked in with the `$R` directive. This is a very flexible way to store a wizard's source code so that it can be readily modified. The `RES` file is built by including a text file and `RCDATA` resource in an `RC` file and then compiling that `RC` file with `BRCC32`. Listings 17.10 and 17.11 show the contents of `CodeGen.txt` and `CodeGen.rc`.

LISTING 17.10 CodeGen.txt—the Resource Template for the AppBar Wizard

```
unit %0:s;

interface

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

type
  T%1:s = class(%2:s)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  %1:s: T%1:s;

implementation

{$R *.DFM}

end.
```

LISTING 17.11 CODEGEN.RC

```
CODEGEN RCDATA CODEGEN.TXT
```

Registration of the custom module and wizard occurs inside of a `Register()` procedure in the design package containing the wizard using the following two lines:

```
RegisterCustomModule(TAppBar, TCustomModule);
RegisterPackageWizard(TAppBarWizard.Create);
```

Summary

After reading this chapter, you should have a greater understanding of the various units and interfaces involved in the Delphi Open Tools API. In particular, you should know and understand the issues involved in creating wizards that plug into the IDE. This chapter completes the "Component-Based Development" section of the book. In the next section, "Enterprise Development," you will learn techniques for building enterprise-grade applications, starting with those based on COM+ and MTS.