# DDG Bug-Reporting Tool: Desktop Application Development

CHAPTER

35

## IN THIS CHAPTER

This chapter discusses techniques for developing desktop database applications. The DDG bug-reporting application illustrates several methods to take into consideration, in particular, how to design an application that may be deployed to the Internet. In this demo, we also illustrate several techniques and tricks to get around some sticky issues when separating the user interface from the data-manipulation routines.

Because of Delphi's ease of use, developing database applications is simple. However, it is also easy to overlook issues that may end up biting you later when you want to extend the application's basic functionality. In this chapter, we will show you how to take this into account when creating your database applications.

# General Application Requirements

The general requirements for the DDG bug-reporting application are discussed in this section. Be aware that our intentions were not to actually design a deployable bug-reporting tool. Rather, we use a real-world need to illustrate the techniques discussed in this chapter. Therefore, we left out functionality that you might expect from this application so as not to cloud our techniques with application logic.

## World Wide Web–Ready

The bug-reporting application must be designed in such a manner as to minimize the development effort to make its functionality available on the World Wide Web. This means that the user interface must be completely—not almost completely—separated from the database logic. In essence, you should be able to attach different user interfaces to the database logic. In fact, you will see this in Chapter 36, "DDG Bug-Reporting Tool: Using WebBroker," when we make our application available through Web pages.

## User Data Entry and Logon

The bug-reporting application contains a table of users who can log on to the system. These users can report the existence of bugs by using this application. Users can also add other users to the bug-reporting application. For this version of the application, it is not required that the users be able to edit or delete user information.

A user logs on to the bug-reporting tool by providing a user name, which is stored in the `Users.db` table. This logon is only for the process of obtaining the user ID, which is needed to manipulate reported bugs—this is not a security measure.

## Bug Manipulation, Browsing, and Filtering

Users can add, edit, and delete bug information. Users can also provide the necessary field information for each bug. For example, a user can enter the date the bug is reported; assign it

to another user; and specify a status, a summary, details, and the affected source. The user entering the bug is added automatically.

### Bug Actions

Users can add actions (notes) to an existing bug report. Users can also browse actions previously entered by themselves or by other users. This is a handy way to track the bug-correction progress and for interested parties to pass notes back and forth about a bug.

### Other UI Functionality

The application must make use of techniques necessary to make the user interface easy to understand and use. Features such as lookup fields and "friendly" display labels will be used where necessary.

## The Data Model

The data model for the bug-reporting application is shown in Figure 35.1. Here is what the tables in the model consist of:

- `IDs`. The `IDs` table serves as the key generation table and keeps track of the next available key for the `Users`, `Bugs`, and `Actions` tables.
- `Users`. The `Users` table stores users who are added to the bug-reporting system.
- `Bugs`. The `Bugs` table stores the general information about bugs.
- `Actions`. The `Actions` table stores notes on bugs. Each bug may have several notes.
- `Status`. The `Status` table is a lookup table for assigning a specific status to each bug.

## Developing the Data Module

The data module is the central piece to the bug-reporting application. It is through the data module that all database manipulation is handled. The user interface uses the data module's functionality through public methods and properties. No direct reference to data-access components is made from any user interface element except where necessary from the Object Inspector. An example of directly accessing a data-access component would be in the `DataSet` property for the `TDataSource` component that resides on UI forms. Likewise, and even more important, the data module should never access elements that reside in the user interface.

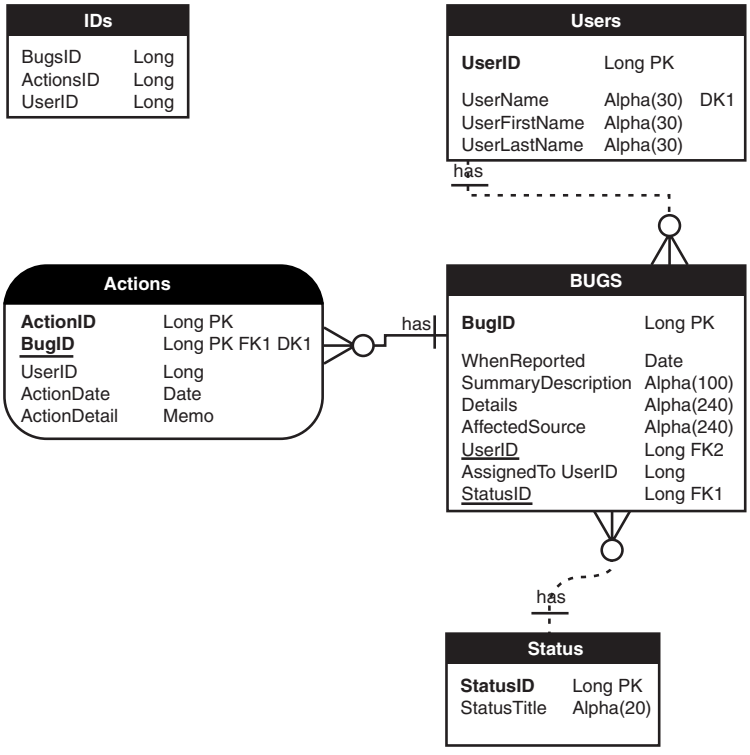


**FIGURE 35.1**
*The bug-reporting application data model.*

**NOTE**

When developing applications in which you want to separate data logic from the user interface, the placement of the `TDataSource` component is not of grave concern. We chose to place it on the UI forms rather than in the data module because we feel it has more to do with user interface than data access. This, however, is a preference, and you may choose to do otherwise for whatever reason.

Listing 35.1 shows the source code for the bug application's data module.

**LISTING 35.1** Data Module for the DDGBugs Application

```
unit DDGBugsDM;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Db, DBTables, HTTPApp, DBWeb;

type

  EUnableToObtainID = class(Exception);

  TDDGBugsDataModule = class(TDataModule)
    dbDDGBugs: TDatabase;
    tblBugs: TTable;
    tblUsers: TTable;
    tblStatus: TTable;
    tblActions: TTable;
    tblBugsBugID: TIntegerField;
    tblBugsWhenReported: TDateField;
    tblBugsSummaryDescription: TStringField;
    tblBugsDetails: TStringField;
    tblBugsAffectedSource: TStringField;
    tblBugsUserID: TIntegerField;
    tblBugsStatusID: TIntegerField;
    dsUsers: TDataSource;
    dsStatus: TDataSource;
    tblIDs: TTable;
    tblBugsUserNameLookup: TStringField;
    tblBugsAssignedToLookup: TStringField;
    tblUsersUserID: TIntegerField;
    tblUsersUserName: TStringField;
    tblUsersUserFirstName: TStringField;
    tblUsersUserLastName: TStringField;
    tblBugsAssignedToUserID: TIntegerField;
    dsBugs: TDataSource;
    wbdpBugs: TWebDispatcher;
    dstpBugs: TDataSetTableProducer;
    procedure DDGBugsDataModuleCreate(Sender: TObject);
    procedure tblBugsBeforePost(DataSet: TDataSet);
    procedure tblBugsFilterRecord(DataSet: TDataSet; var Accept: Boolean);
    procedure tblUsersBeforePost(DataSet: TDataSet);
    procedure tblBugsAfterInsert(DataSet: TDataSet);
    procedure wbdpBugswaShowAllBugsAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure wbdpBugswaIntroAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure wbdpBugswaUserNameAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure wbdpBugswaVerifyUserNameAction(Sender: TObject;
```

*continues*

**LISTING 35.1** Continued

```
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
private
  FLoginUserID: Integer;
  FLoginUserName: String;

  function GetFilterOnUser: Boolean;
  procedure SetFilterOnUser(const Value: Boolean);
  function GetNumBugs: Integer;
protected
  procedure PostAction(Sender: TObject; Action: TStrings);
public

  // Bugs Methods
  procedure FirstBug;
  procedure LastBug;
  procedure NextBug;
  procedure PreviousBug;
  function IsLastBug: Boolean;
  function IsFirstBug: Boolean;
  function IsBugsTblEmpty: Boolean;
  procedure InsertBug;
  procedure DeleteBug;
  procedure EditBug;
  procedure SaveBug;
  procedure CancelBug;
  procedure SearchForBug;

  // User Functions
 {$IFNDEF DDGWEBBUGS}
  procedure AddUser;
 {$ENDIF}

  procedure PostUser(Sender: TObject);
  function GetUserFLName(AUserID: Integer): String;

  // Action Methods

  {$IFNDEF DDGWEBBUGS}
  procedure AddAction;
  {$ENDIF}

  procedure GetActions(AActions: TStrings);

  // Id Generation
  function GetDataSetID(const AFieldName: String): Integer;
```

```
    function GetNewBugID: Integer;
    function GetNewUserID: Integer;
    function GetNewActionID: Integer;

    // Login Function
    function Login: Boolean;

    // Exposed properties

    property LoginUserID: Integer read FLoginUserID;
    property FilterOnUser: Boolean read GetFilterOnUser write SetFilterOnUser;
    property NumBugs: Integer read GetNumBugs;
  end;

var
  DDGBugsDataModule: TDDGBugsDataModule;

implementation

{$IFNDEF DDGWEBBUGS}
uses UserFrm, ActionFrm;
{$ENDIF}

{$R *.DFM}

// Helper functions.

function IsInteger(IntVal: String): Boolean;
var
  v, code: Integer;
begin
  val(IntVal, v, code);
  Result := code = 0;
end;

procedure MemoFromStrings(AMemoField: TMemoField; AStrings: TStrings);
var
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    AStrings.SaveToStream(Stream);
    Stream.Seek(0, soFromBeginning);
    AMemoField.LoadFromStream(Stream);
  finally
    Stream.Free;
```

*continues*

**LISTING 35.1** Continued

```
  end;
end;

procedure StringsFromMemo(AStrings: TStrings; AMemoField: TMemoField);
var
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    AMemoField.SaveToStream(Stream);
    Stream.Seek(0, soFromBeginning);
    AStrings.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
end;

// Internal methods

function TDDGBugsDataModule.GetFilterOnUser: Boolean;
begin
  Result := tblBugs.Filtered;
end;

procedure TDDGBugsDataModule.SetFilterOnUser(const Value: Boolean);
begin
 tblBugs.Filtered := Value;
end;

function TDDGBugsDataModule.GetNumBugs: Integer;
begin
  Result := tblBugs.RecordCount;
end;

// Identifier methods

function TDDGBugsDataModule.GetDataSetID(const AFieldName: String): Integer;
const
  MaxAttempts = 50;
var
  Attempts: Integer;
  NextID: Integer;
begin
  tblIDs.Active := True;
  // Try fifty times if until this works or raise an exception
```

```
  Attempts := 0;
  while Attempts <= MaxAttempts do
  begin
    try
      Inc(Attempts);
      // If another user has the table in edit mode, an error occurs here.
      tblIDs.Edit;
      // If we reach the Break statement, we are successful. Break out of loop.
      Break;
    except
      on EDBEngineError do
      begin
        // Do some delay
        Continue;
      end;
    end;
  end;

  if tblIDs.State = dsEdit then
  begin
    // Increment the value obtained from the table and restore the new value
    // to the table for the next record.
    NextID := tblIDs.FieldByName(AFieldName).AsInteger;
    tblIDs.FieldByName(AFieldName).AsInteger := NextID + 1;
    TblIDs.Post;
    Result := NextID;
  end
  else
    Raise EUnableToObtainID.Create('Cannot create unique ID');
end;

function TDDGBugsDataModule.GetNewActionID: Integer;
begin
  Result := GetDataSetID('ActionsID');
end;

function TDDGBugsDataModule.GetNewBugID: Integer;
begin
  Result := GetDataSetID('BugsID');
end;

function TDDGBugsDataModule.GetNewUserID: Integer;
begin
  Result := GetDataSetID('UsersID');
end;
```

*continues*

**LISTING 35.1** Continued

```
// Initialization/Login methods.

procedure TDDGBugsDataModule.DDGBugsDataModuleCreate(Sender: TObject);
begin
  { These tables are opened in the proper order so the master-detail
    relationship does not fail.}
  dbDDGBugs.Connected := True;
  tblUsers.Active  := True;
  tblStatus.Active := True;
  tblBugs.Active   := True;
  tblActions.Active := True;
end;

function TDDGBugsDataModule.Login: Boolean;
var
  UserName: String;
begin
  InputQuery('Login', 'Enter User Name: ', UserName);
  Result := tblUsers.Locate('UserName', UserName, []);
  if Result then
  begin
    FLoginUserID   := tblUsers.FieldByName('UserID').AsInteger;
    FLoginUserName := tblUsers.FieldByName('UserName').AsString;
  end;
end;

// Bug methods.

procedure TDDGBugsDataModule.FirstBug;
begin
  tblBugs.First;
end;

procedure TDDGBugsDataModule.LastBug;
begin
  tblBugs.Last;
end;

procedure TDDGBugsDataModule.NextBug;
begin
  tblBugs.Next;
end;

procedure TDDGBugsDataModule.PreviousBug;
begin
```

```
  tblBugs.Prior;
end;

function TDDGBugsDataModule.IsLastBug: Boolean;
begin
  Result := tblBugs.Eof;
end;

function TDDGBugsDataModule.IsFirstBug: Boolean;
begin
  Result := tblBugs.Bof;
end;

function TDDGBugsDataModule.IsBugsTblEmpty: Boolean;
begin
  // If RecordCount is zero, there are not bugs in the table.
  Result := tblBugs.RecordCount = 0;
end;

procedure TDDGBugsDataModule.InsertBug;
begin
  tblBugs.Insert;
end;

procedure TDDGBugsDataModule.DeleteBug;
var
  Qry: TQuery;
  BugID: Integer;
begin
  if MessageDlg('Delete Action?', mtConfirmation,
     [mbYes, mbNo], 0) = mrYes then
  begin
    BugID := tblBugs.FieldByName('BugID').AsInteger;
    // Use a dynamically created TQuery component to perform these operations.
    Qry := TQuery.Create(self);
    try
      dbDDGBugs.StartTransaction;
      try
        // First delete any action belonging to this bug.
        Qry.DatabaseName := dbDDGBugs.DataBaseName;
        Qry.SQL.Add(Format('DELETE FROM ACTIONS WHERE BugID = %d', [BugID]));
        Qry.ExecSQL;

        // Now delete bug from the bugs table.
        Qry.SQL.Clear;
        Qry.SQL.Add(Format('DELETE FROM BUGS WHERE BugID = %d', [BugID]));
```

*continues*

**LISTING 35.1** Continued

```
        Qry.ExecSQL;

        tblBugs.Refresh;
        tblActions.Refresh;

        dbDDGBugs.Commit;
      except
        dbDDGBugs.Rollback;
        raise;
      end;
    finally
      Qry.Free;
    end;
  end;
end;

procedure TDDGBugsDataModule.EditBug;
begin
  tblBugs.Edit;
end;

procedure TDDGBugsDataModule.SaveBug;
begin
  tblBugs.Post;
end;

procedure TDDGBugsDataModule.CancelBug;
begin
  tblBugs.Cancel;
end;

procedure TDDGBugsDataModule.SearchForBug;
var
  BugStr: String;
begin
  InputQuery('Search for bug', 'Enter bug ID: ', BugStr);
  if IsInteger(BugStr) then
    if not tblBugs.Locate('BugID', StrToInt(BugStr), []) then
      MessageDlg('Bug not found.', mtInformation, [mbOK], 0);
end;

// User methods.

{$IFNDEF DDGWEBBUGS}
procedure TDDGBugsDataModule.AddUser;
```

```
begin
  tblUsers.Insert;
  try
    if NewUserForm(PostUser) = mrCancel then
      tblUsers.Cancel;
  except
    { An error occurred. Put the table to browse mode
        and reraise the exception }
    tblUsers.Cancel;
    raise;
  end;
end;
{$ENDIF}

procedure TDDGBugsDataModule.PostUser(Sender: TObject);
begin
  if tblUsers.State = dsInsert then
    tblUsers.FieldByName('UserID').AsInteger := GetNewUserID;
  tblUsers.Post;
end;

function TDDGBugsDataModule.GetUserFLName(AUserID: Integer): String;
begin
  // Returns the first and last name concatenated.
  if tblUsers.Locate('UserID', AUserID, []) then
    Result := Format('%s %s', [tblUsers.FieldByName('UserFirstName').AsString,
      tblUsers.FieldByName('UserLastName').AsString])
  else
    Result := EmptyStr;
end;

{$IFNDEF DDGWEBBUGS}
procedure TDDGBugsDataModule.AddAction;
begin
  NewActionForm(PostAction);
end;
{$ENDIF}

procedure TDDGBugsDataModule.GetActions(AActions: TStrings);
var
  Action: TStringList;
  ActionUserId: Integer;

begin
  Action := TStringList.Create;
  try
```

*continues*

**LISTING 35.1** Continued

```
    with tblActions do
    begin
      tblActions.First;
      while not Eof do
      begin
        Action.Clear;
        ActionUserID := FieldByName('UserID').AsInteger;
        StringsFromMemo(Action, TMemoField(FieldByName('ActionDetail')));
        AActions.Add(Format('Action Added on: %s',
             [FormatDateTime('mmm dd, yyyy',
          FieldByName('ActionDate').AsDateTime)]));
        AActions.Add(Format('Action Added by: %s',
             [GetUserFLName(ActionUserID)]));
        AActions.Add(EmptyStr);
        AActions.AddStrings(Action);
        AActions.Add('============================');
        AActions.Add(EmptyStr);
        tblActions.Next;
      end; // while
    end; // with
  finally
    Action.Free;
  end;
end;

procedure TDDGBugsDataModule.PostAction(Sender: TObject; Action: TStrings);
var
  BugID: Integer;
begin
  tblActions.Insert;
  try
    BugID := tblBugs.FieldByName('BugID').AsInteger;
    tblActions.FieldByName('ActionID').AsInteger      := GetNewActionID;
    tblActions.FieldByName('BugID').AsInteger         := BugID;
    tblActions.FieldByName('UserID').AsInteger        := LoginUserID;
    tblActions.FieldByName('ActionDate').AsDateTime   := Date;
    MemoFromStrings(TMemoField(tblActions.FieldByName('ActionDetail')),
      Action);
    tblActions.Post;
  except
    tblActions.Cancel;
    raise;
  end;
end;
```

```
// Event Handlers

procedure TDDGBugsDataModule.tblBugsBeforePost(DataSet: TDataSet);
begin
  if tblBugs.State = dsInsert then
    tblBugs.FieldByName('BugID').AsInteger := GetNewBugID;
end;

procedure TDDGBugsDataModule.tblBugsFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept := tblBugs.FieldByName('UserID').AsInteger = FLoginUserID;
end;

procedure TDDGBugsDataModule.tblUsersBeforePost(DataSet: TDataSet);
begin
  if tblUsers.State = dsInsert then
    tblUsers.FieldByName('UserID').AsInteger := GetNewUserID;
end;

procedure TDDGBugsDataModule.tblBugsAfterInsert(DataSet: TDataSet);
begin
  tblBugs.FieldByName('UserID').AsInteger := FLoginUserID;
  tblBugs.FieldByName('UserNameLookup').AsString := FLoginUserName;
end;

end.
```

## Application Initialization and Login

You will see in Listing 35.2 that we moved the `TDDGBugsDataModule` so that it is created first. Then we call its `Login()` method, which determines whether the application execution continues. It makes this determination based on whether the username entered actually exists in the `Users.db` table, as shown in the `TDDGBugsDataModule.Login()` method in Listing 35.1.

In order to support user logins, we had to modify the project file as shown in Listing 35.2.

**LISTING 35.2** Project File for the Bug-Reporting Application

```
program DDGBugs;

uses
  Forms,
  Dialogs,
  ChildFrm in '..\ObjRepos\CHILDFRM.pas' {ChildForm},
```

*continues*

**LISTING 35.2** Continued

```
  DBModeFrm in '..\ObjRepos\DBMODEFRM.pas' {DBModeForm},
  DBNavStatFrm in '..\ObjRepos\DBNAVSTATFRM.pas' {DBNavStatForm},
  MainFrm in 'MainFrm.pas' {MainForm},
  UserFrm in 'UserFrm.pas' {UserForm},
  ActionFrm in 'ActionFrm.pas' {ActionForm},
  DDGBugsDM in '..\Shared\DDGBugsDM.pas' {DDGBugsDataModule: TDataModule};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TDDGBugsDataModule, DDGBugsDataModule);
  if DDGBugsDataModule.Login then
  begin
    Application.CreateForm(TMainForm, MainForm);
    Application.Run;
  end
  else
    MessageDlg('Invalid Login', mtError, [mbOk], 0);
end.
```

## Generating Paradox Keys

Because the bug-reporting application uses the Paradox database as the back end, we acquire a slight anomaly that we must get around. This anomaly has to do with the Paradox autoincrement fields. Although Paradox's autoincrement fields can supposedly allow you to use them as key fields, they are highly unreliable. Our experience has been that they can easily become out of sync with foreign keys. We opted to avoid their use and create our own keys based on the values contained in the `IDs.db` table.

The `IDs.db` table stores the next available integer value for the Bugs, Users, and Action keys. The `TDDGBugsDataModule.GetDataSetID()` method ensures that only one user is able to put the specific key field of the `tblIDs` table into Edit mode. This will ensure that no two users get identical key values when inserting records. `GetDataSetID()` is made generic for the three types of keys by passing in the field name for the key value desired. Therefore, this method can be used to obtain keys for bugs, users, and actions. In fact, this method is called by the `GetNewActionID()`, `GetNewBugID()`, and `GetNewUserID()` methods. These three methods may be called whenever posting a record to one of these tables. You do so in the `BeforePost` event handlers for the `tblBugs` and `tblUsers` tables and in the `PostAction()` method for the `tblActions` table.

## Bug-Manipulation Routines

The bug-manipulation routines are those methods declared under the comment `// Bug Methods`. Most of these functions are self-explanatory—especially the navigation method, which we will not go into. The method `DeleteBug()` contains most of the code for the bug manipulation routines. This method ensures that any actions belonging to a bug get deleted before the bug record is deleted. We will discuss actions shortly. Here, we are using the transaction functionality of `TDatabase` to wrap this operation in a transaction. This will ensure that no data is lost if an error occurs. Note that to perform transaction processing against a local database such as Paradox, you must set the `TransIsolation` property of the `TDatabase` component to `tiDirtyRead`, as we have done.

## Browsing/Filtering Bugs

The user is able to browse all bugs in the database or just those bugs belonging to him or her. This is made possible through the use of the `Filtered` property of the `tblBugs` component. When `tblBugs.Filtered` is `True`, the `tblBugs.OnFilterRecord` property is invoked for each record. Here, you display a record only if its `UserID` field is that of the user logged on, as indicated by the global field `FLoginUserID`. Notice how you surface the `Filtered` property of the `tblBugs` table to the user interface. Instead of allowing the user interface to directly access the `tblBugs.Filtered` property, you surface this property through the `TDDGBugsDataModule.FilterOnUser` property. This property's writer method, `SetFilterOnUser()`, performs the assignment to the `tblBugs.Filtered` property. Now, you cannot actually enforce the rule that forms cannot directly access properties of components that reside on a `TDatamodule` because the VCL is not using strict OOP visibility rules.

## Adding Users

Adding users is done through the `TDDGBugsDataModule.AddUser()` and `TDDGBugsDataModule.PostUser()` methods. The `AddUser()` method invokes a simple dialog with which you add the user data. Note how the `PostUser()` method is passed to the `NewUserForm()` function, which invokes the user form. This illustrates how you can avoid having to make a form invoked by a data module refer right back to that data module. The reason this problem presented itself is because we are protecting the data module components from external access. There are probably a number of ways we could have accomplished this—this just happens to be the one we chose. `NewUserForm()` invokes the form defined in the `UserFrm.pas` unit shown in Listing 35.3.

**LISTING 35.3** UserFrm.pas: The User F30orm

```
unit UserFrm;

interface

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

type

  TUserForm = class(TForm)
    lblUserName: TLabel;
    dbeUserName: TDBEdit;
    lblFirstName: TLabel;
    dbeFirstName: TDBEdit;
    lblLastName: TLabel;
    dbeLastName: TDBEdit;
    btnOK: TButton;
    btnCancel: TButton;
    procedure btnOKClick(Sender: TObject);
  private
    FPostUser: TNotifyEvent;
  public
    { Public declarations }
  end;

function NewUserForm(APostUser: TNotifyEvent): Word;

implementation
uses dbTables;
{$R *.DFM}

function NewUserForm(APostUser: TNotifyEvent): Word;
var
  UserForm: TUserForm;
begin
  UserForm := TUserForm.Create(Application);
  try
    UserForm.FPostUser := APostUser;
    Result := UserForm.ShowModal;
  finally
    UserForm.Free;
  end;
end;

procedure TUserForm.btnOKClick(Sender: TObject);
```

```
begin
  if dbeUserName.Text = EmptyStr then begin
    MessageDlg('A user name is required.', mtWarning, [mbOK], 0);
    dbeUserName.SetFocus;
    ModalResult := mrNone;
  end
  else begin
    try
      FPostUser(self);
    except
      on EDBEngineError do
      begin
        MessageDlg('User name already exists.', mtWarning, [mbOK], 0);
        dbeUserName.SetFocus;
        ModalResult := mrNone;
      end;
    end;
  end;
end;

end.
```

As shown in Listing 35.3, `NewUserForm()` creates the `TUserForm` and displays it. Notice that you assign the `APostUser` parameter to the `FPostUser` field that is of the type `TNotifyEvent`. By declaring `FPostUser` as a method pointer (`TNotifyEvent`), you can assign the `PostUser()` method from `TDDGBugDataModule` to `FPostUser` because `PostUser()` matches the definition of `TNotifyEvent`. This concept was covered in Chapters 20, "Key Elements of the VCL and Runtime Type Information," and 21, "Writing Delphi Custom Components."

When the user clicks the OK button, `btnOkClick()` is invoked. Provided a user name was entered, the `TDDGBugDataModule.PostUser()` method (referred to by `FPostUser`) is invoked, which should save the user record (see `PostUser()` in Listing 35.1). If an error occurs in `PostUser()`, the user name already exists in the database. This illustrates another advantage to passing the `PostUser()` method to the `TUserForm`. The `TUserForm` can handle an error raised in the data module. This concept is not that different from developing components. You develop the data module such that it is completely self-contained. You also allow users of the data module to handle any errors raised within the data module.

## Adding Actions

Actions are basically notes that are optionally attached to each bug. Anybody can add an action to a bug. The `TDDGBugsDataModule.AddAction()` method calls `NewActionForm()`, which obtains the action data from the user and adds it to the database. `NewActionForm()` is defined in `ActionFrm.pas`, which is shown in Listing 35.4.

**LISTING 35.4** ActionFrm.pas: The Action Form

```
unit ActionFrm;

interface

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

type

  TPostActionEvent = procedure (Sender: TObject; Action: TStrings) of Object;

  TActionForm = class(TForm)
    memAction: TMemo;
    lblAction: TLabel;
    btnOK: TButton;
    btnCancel: TButton;
    procedure btnOKClick(Sender: TObject);
  private
    FPostAction: TPostActionEvent;
  public
    { Public declarations }
  end;


procedure NewActionForm(APostAction: TPostActionEvent);

implementation
{$R *.DFM}

procedure NewActionForm(APostAction: TPostActionEvent);
var
  ActionForm: TActionForm;
begin
  ActionForm := TActionForm.Create(Application);
  try
    ActionForm.FPostAction := APostAction;
    ActionForm.ShowModal;
  finally
```

```
    ActionForm.Free;
  end;
end;

procedure TActionForm.btnOKClick(Sender: TObject);
begin
  if Assigned(FPostAction) then
    FPostAction(Self, memAction.Lines);
end;

end.
```

Much like `NewUserForm()`, the `NewActionForm()` method takes a method pointer as a parameter. This time, we defined our own method type of `TPostActionEvent`, which takes a `TObject` and the `TStrings` object containing the action text. When the user clicks the OK button, the `btnOKClick()` event is invoked, which in turn invokes `TDDGBugsDataSource.PostAction()` to add the action to the database (`FPostAction` refers to `PostAction()`).

You can refer to the source commentary for additional information on the data module. Later, you will see how to add code to this data module to share it with another application—an ISAPI server that Web-enables the bug program.

# Developing the User Interface

In this section, we will discuss the development of the user interface for this application. We will also point out some preparations you can make for Web deployment of this application.

## The Main Form

The user interface basically refers to the methods of the data module. We have a single form interface consisting of three pages. The first page allows the user to add, edit, and view the bug information. The second page is for browsing actions. The third page allows the user to view a grid that contains either all the bugs or only the currently logged-in user's bugs. Figures 35.2, 35.3, and 35.4 show the three pages for the main form.

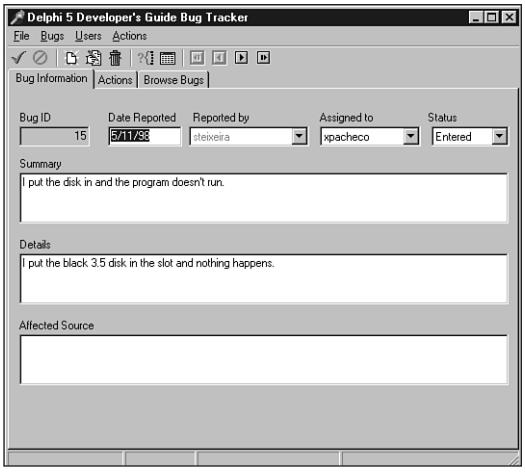


**FIGURE 35.2**
*The Bug Information page.*

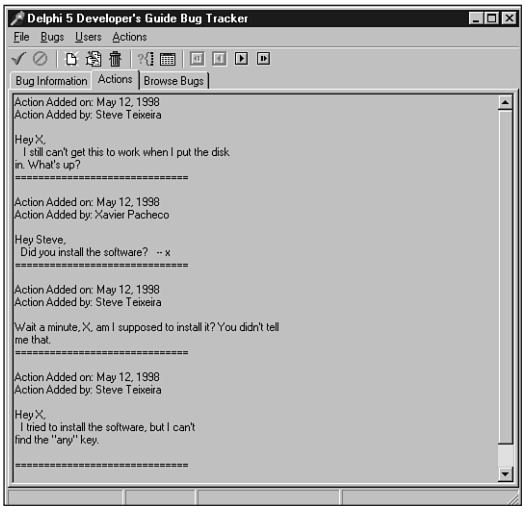


**FIGURE 35.3**
*The Actions page.*

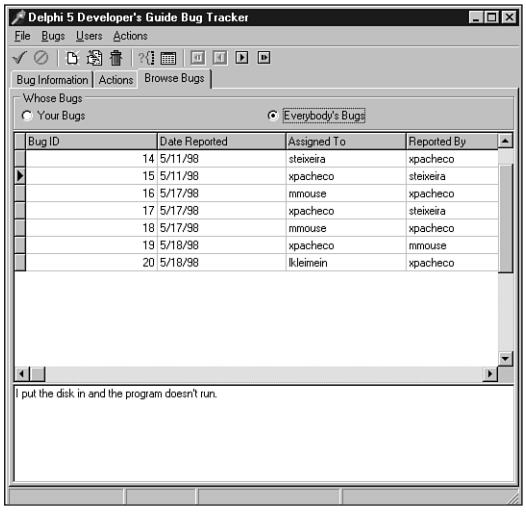


**FIGURE 35.4**
*The Browse Bugs page.*

`TMainForm` is defined in `MainFrm.pas`, which is shown in Listing 35.5.

**LISTING 35.5** The Main Form for the DDGBugs Application

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBNAVSTATFRM, Menus, ImgList, ComCtrls, ToolWin, StdCtrls, DBCtrls, Db,
  Mask, dbModeFrm, ActnList, Grids, DBGrids, ExtCtrls;

type

  TMainForm = class(TDBNavStatForm)
    pcMain: TPageControl;
    tsBugInformation: TTabSheet;
    tsActions: TTabSheet;
    lblBugID: TLabel;
    dbeBugID: TDBEdit;
    dsBugs: TDataSource;
    lblDateReported: TLabel;
    lblSummary: TLabel;
```

*continues*

**LISTING 35.5** Continued

```
    lblDetails: TLabel;
    lblAffectedSource: TLabel;
    lblReportedBy: TLabel;
    lblAssignedTo: TLabel;
    lblStatus: TLabel;
    dbmSummary: TDBMemo;
    dbmDetails: TDBMemo;
    dbmAffectedSource: TDBMemo;
    tsBrowseBugs: TTabSheet;
    rgWhoseBugs: TRadioGroup;
    dbgBugs: TDBGrid;
    dbmSummary2: TDBMemo;
    memAction: TMemo;
    dblcAssignedTo: TDBLookupComboBox;
    dblcStatus: TDBLookupComboBox;
    dbeDateReported: TDBEdit;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiUsers: TMenuItem;
    mmiAddUser: TMenuItem;
    mmiActions: TMenuItem;
    mmiAddActionToBug: TMenuItem;
    dblcReportedBy: TDBLookupComboBox;
    procedure FormCreate(Sender: TObject);
    procedure sbFirstClick(Sender: TObject);
    procedure sbPreviousClick(Sender: TObject);
    procedure sbNextClick(Sender: TObject);
    procedure sbLastClick(Sender: TObject);
    procedure sbSearchClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure sbAcceptClick(Sender: TObject);
    procedure sbCancelClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
    procedure sbDeleteClick(Sender: TObject);
    procedure sbBrowseClick(Sender: TObject);
    procedure rgWhoseBugsClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiAddUserClick(Sender: TObject);
    procedure mmiAddActionToBugClick(Sender: TObject);
    procedure dsBugsDataChange(Sender: TObject; Field: TField);
  private
    procedure SetActionStatus;

  protected
```

```
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

uses DDGBugsDM;

{$R *.DFM}

{ TMainForm }

procedure TMainForm.SetActionStatus;
begin
  mmiFirst.Enabled := not DDGBugsDataModule.IsFirstBug;
  mmiLast.Enabled  := not DDGBugsDataModule.IsLastBug;
  mmiNext.Enabled  := not DDGBugsDataModule.IsLastBug;
  mmiPrevious.Enabled := not DDGBugsDataModule.IsFirstBug;
  mmiDelete.Enabled := not DDGBugsDataModule.IsBugsTblEmpty;

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

  // User cannot add users or actions when adding/editing a bug.
  mmiUsers.Enabled := FormMode = fmBrowse;
  mmiActions.Enabled := (FormMode = fmBrowse) and
     (DDGBugsDataModule.NumBugs <> 0);

  { disable the browsing of bug records when the user is editing or adding
    a new bug. }
  dbgBugs.Enabled     := FormMode = fmBrowse;
  rgWhoseBugs.Enabled := FormMOde = fmBrowse;

end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  inherited;
  SetActionStatus;
end;
```

*continues*

**LISTING 35.5** Continued

```
procedure TMainForm.sbFirstClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.FirstBug;
  SetActionStatus;
end;

procedure TMainForm.sbPreviousClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.PreviousBug;
  SetActionStatus;
end;

procedure TMainForm.sbNextClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.NextBug;
  SetActionStatus;
end;

procedure TMainForm.sbLastClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.LastBug;
  SetActionStatus;
end;

procedure TMainForm.sbSearchClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.SearchForBug;
end;

procedure TMainForm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var
  Rslt: word;
begin
  inherited;
  if not (FormMode = fmBrowse) then
  begin
    rslt := MessageDlg('Save changes?', mtConfirmation, mbYesNoCancel, 0);
    case rslt of
      mrYes:
        begin
          DDGBugsDataModule.SaveBug;
```

```
          FormMode := fmBrowse;
          CanClose := True;
        end;
      mrNo:
        begin
          DDGBugsDataModule.CancelBug;
          FormMode := fmBrowse;
          CanClose := True;
        end;
      mrCancel:
        CanClose := False;
    end;
  end;
end;

procedure TMainForm.sbAcceptClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.SaveBug;
  SetActionStatus;
end;

procedure TMainForm.sbCancelClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.CancelBug;
  SetActionStatus;
end;

procedure TMainForm.sbInsertClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.InsertBug;
  SetActionStatus;
end;

procedure TMainForm.sbEditClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.EditBug;
  SetActionStatus;
end;

procedure TMainForm.sbDeleteClick(Sender: TObject);
begin
  inherited;
```

*continues*

**LISTING 35.5** Continued

```
  DDGBugsDataModule.DeleteBug;
  SetActionStatus;
end;

procedure TMainForm.sbBrowseClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.CancelBug;
  SetActionStatus;
end;

procedure TMainForm.rgWhoseBugsClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.FilterOnUser := rgWhoseBugs.ItemIndex = 0;
end;

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

procedure TMainForm.mmiAddUserClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.AddUser;
end;

procedure TMainForm.mmiAddActionToBugClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.AddAction;
  dsBugsDataChange(nil, nil);
end;

procedure TMainForm.dsBugsDataChange(Sender: TObject; Field: TField);
begin
  inherited;
  // A new bug is being displayed so clear the action list and
  // retrieve the actions for the newly displayed bug.
  memAction.Lines.Clear;
  DDGBugsDataModule.GetActions(memAction.Lines);
end;

end.
```

TMainForm descends from TDBNavStatForm, which was presented back in Chapter 4, "Application Frameworks and Design Concepts." It should exist in your Object Repository. TDBNavStatForm contains the functionality to update the speedbuttons and status bar based on the form's mode (Add, Edit, or Browse). Most of the methods simply call the corresponding data module methods.

Note that you set the dsBugs.AutoEdit property to False, thus preventing the user from inadvertently placing the table into Edit mode. You want to have the user explicitly set the Bugs table to Edit or Insert mode by clicking the appropriate buttons or menu items.

This form is uncomplicated. SetActionStatus() simply enables/disables buttons and menus based on various conditions. FormCloseQuery() ensures that the user saves or cancels any pending edit or insert.

## Other User Interface Issues

From the data module, we have controlled how field labels are displayed by adding the fields to the TTable object and specifying a friendlier label in the Object Inspector. The same can be done for TDBGrid objects by modifying the Title property of the TDBGrid.Columns property. We have used both methods to control the labels displayed to the user.

# Enabling the Application for the Web

We stated earlier that a Web-enabled version of the application is required. To make this possible, we need to remove any references to any forms from within TDDGBugsDataModule. Using the conditional compilation directives you will see in the DDGBugsDM.pas unit does this. For example, examine the following code:

```
{$IFNDEF DDGWEBBUGS}
 procedure AddUser;
{$ENDIF}
```

The {$IFNDEF} condition ensures that the AddUser() method is compiled into the application only if the DDBWEBBUGS conditional directive is not defined, which is the case for this application.

# Summary

In this chapter we discussed techniques for developing a desktop database application. We also emphasized separating the user interface from the data manipulation routines. This will make converting the application to a Web-enabled version easier. The next chapter, "DDG Bug-Reporting Tool: Using WebBroker," illustrates how to do just that.