Large Database applications in Delphi - Part 3: Extending TForm

March 16, 2014

Abstract

In this series of articles, some programming ideas will be discussed that can be applied when making large database applications in Delphi, i.e. database applications that have many screens, which operates on a database with possibly many tables. Many aspects of such programs will be discussed; No code will be presented other than some small code snippets to illustrate the ideas.

1 Introduction

In this article several ways of extending `TForm` will be presented: In large projects with many screens, there is a lot of common functionality in the screens, especially if the application is a MDI application or a Database applicion. Some of these functionality may be specific to a project, e.g. some logging of user actions. Other functionality is common to most applications: e.g. when data was modified in the form and the form is closed, the application should ask wether this data should be saved. Since `TForm` is a class as any other, OOP techniques can be used to implement functionality common to all forms in an ancestor form. This is the idea of form inheritance. In this article this concept will be taken a bit further.

Basically, `TForm`, or better, `TCustomForm` is a small wrapper around a Windows 'window' object. It introduces some properties, some of which are wrappers around the window object (Top, Left, Width Height), others (Action, Scaled) are introduced by Borland to make programming easier.

This means that much functionality which is found in common programs is simply not present in `TForm`. One thing is for instance the automatic saving and restoring of the position and size of a form. For database programs it seems logical to assume a kind of DB-aware form, but neither of these have been implemented by Borland.

There are now 3 ways of continuing:

1. Implement all needed functionality each time in each form. When programming many similar screens, it is obvious that this is an error-prone method, which consumes a lot of time.
2. Use form inheritance as it is standard available in Delphi. This is probably the most practiced way. There are 3 problems with this:
    - When developing in team, this means there is the need for a central repository (called the 'shared repository' in Delphi). This also means that all paths and driveletters must be synchronized on the developer's computers, or the compiler will complain about not found paths.

- When creating a descendent form, the 'parent' form is included in your project by the Delphi IDE. In a modular application as discussed in the first article of this series, this leads to problems; the same unit will be included in all modules, and so only one module can be loaded at a time. (The Delphi IDE suffers from this problem itself; 2 design-time packages that contain a unit with the same name cannot be loaded at the same time)
- `TForm` contains too much properties: If all screens in an application must look the same, then some properties should not be touched by the programmer. It would be better if the property is not there at all, so it cannot be accidentally changed.

3. Implement a new `TForm` which has all needed functionality. This is possible because TForm is just a descendent of `TCustomForm` which publishes all of `TCustomForm`'s properties.

The last approach will be discussed in this article. However, it should be stressed that all things which are discussed here can be done just as well by making an ordinary descendent of `TForm` using standard form inheritance.

2 Starting at `TCustomForm`

When pressing the 'New Form' button in the Delphi IDE, it creates a new `TForm` descendent. What should one do if not a TForm is needed, but a self-made descendent of `TCustomForm` ? And can Delphi handle forms that do not descend from `TForm`, but from `TCustomform` ?

The answer to the latter question is yes, the Delphi IDE can handle `TCustomForms`. The answer to the former question is also simple: Use a wizard which creates the desired descendent of `TCustomForm`. This wizard can be written using the Open Tools API of Delphi.

Luckily, there is no need to write such a wizard from scratch. It has already been written - a long time ago - by Franck Musson (`franck.musson@wanadoo.fr` in a package called rxform50 for Delphi 5 (Entry 14416 in Borland's Code Central). The package can be easily adapted so it compiles with Delphi 6. (The source code on the CD contains a version which compiles under both versions of Delphi).

The author of this article has also extended the package so it can be used to register wizards that create `TDatamodule` descendents.

All that is left to do is to create a `TCustomForm` or `TForm` descendent, and to create a design-time package which registers a wizard to create a custom form.

How can this be done ? In the Delphi IDE, create a new run-time package (it could be named e.g. `AppForms`). Then, in this package, create a new `TForm` (let it be called `TAppForm`), and set all properties as desired. The package should compile.

In Delphi 6, there is a strict separation between run-time packages and design-time packages. Therefore, a second (design-time) package must be made. (e.g. dAppForms). In this second package, which will be included in the Delphi IDE, the `TAppForm` must now be registered in the `rxform` wizard system.

Therefore, a unit is created in the second package (e.g. `regAppForm`). In this unit, a procedure called `Register` should be made. This procedure will be executed when Delphi loads the package (only procedures called Register are executed), and as a result, a new wizard will be installed in Delphi.

The Register procedure should look as follows:

```
Procedure Register;

Const
  AppPage = 'Application Framework';

Var
  AppFormRec : TExpertRecord;

begin
  FillChar(AppFormRec,SizeOf(AppFormRec),0);
  With AppFormRec do
    begin
    Name :=  'App Form';
    Author := 'Michael Van Canneyt';
    Comment := 'App Framework form';
    Page := AppPage;
    Glyph := 0;
    IDString := 'App.AppForm.Form';
    end;
  RegisterCustomFormEx(TAppForm,'frmAppForm', @AppFormRec);
```

For this code to compile, the unit that contains the form (e.g. frmAppForm) should be added to the uses clause of the regAppForm unit, and of course the xptForms50 unit from the rxforms50 package must be added as well.

The `TExpertRecord` is defined in the xptForms50 unit and is used to provide some information about the form which is being registered. The `IDString` field should be unique, it identifies the wizard which will be created in the Delphi IDE.

The `RegisterCustomFormEx` call is also defined in the xptForms50 unit, and does the actual work. The first parameter is the class of the form which should be created; the second parameter is the name of the unit in which the class resides. This name will be added to the `uses` clause of the unit which is created when the wizard is activated. If this name is wrong, the form will not compile. The last parameter is a pointer to a `TExpertRecord` instance.

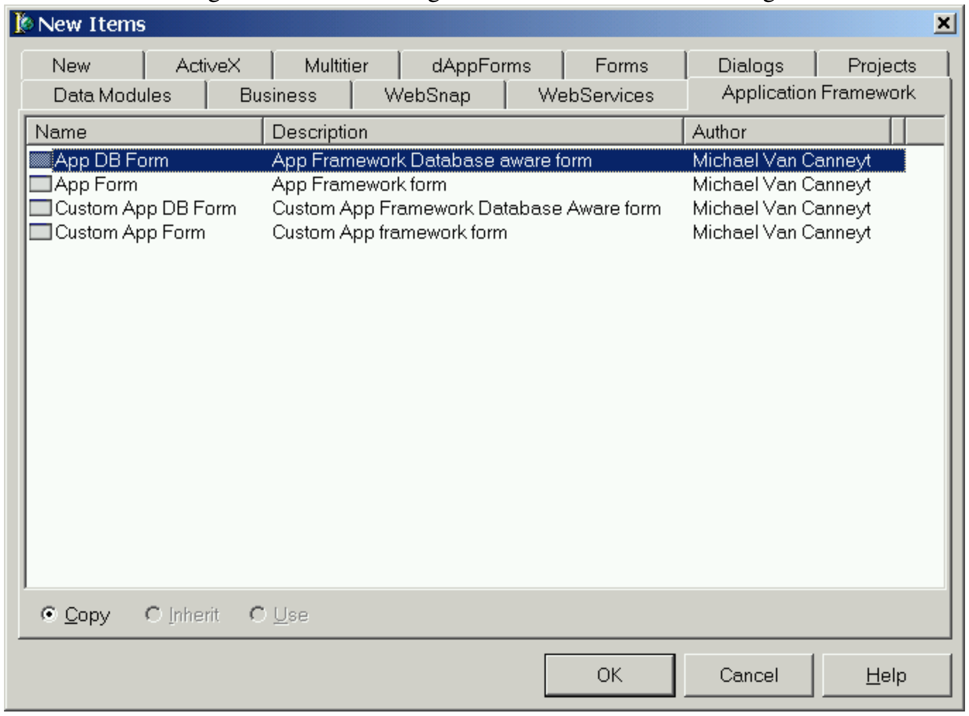
In the demo provided, 4 forms are registered in this way: `TAppForm` and a further descendent `TDBAppForm`. `TAppForm` descends from `TForm`, and `TDBAppForm` descends from `TAppForm`.

The forms could just as well have descended from `TCustomForm`; To illustrate this, two classes `TAppCustomForm` and `TDBAppCustomForm` are registered as well using a similar approach as illustrated above. Note that these classes have *no* form file (with extension .dfm) associated with them, as they are 'non-visual' forms.

Compiling the design-time package and installing it in the Delphi IDE adds a new page to the tabs in the 'New' dialog, called 'Application Framework'. When it is selected, a picture as in figure 1 on page 4 should appear.

Clicking on the desired form will create a new instance of the form, which can be used and designed as any other form. If a `TCustomForm` descendent was made, not all properties may be available. Only the explicitly published properties will be accessible in the IDE object inspector.

Figure 1: The forms registered in the IDE 'New' dialog.



Name	Description	Author
App DB Form	App Framework Database aware form	Michael Van Canneyt
App Form	App Framework form	Michael Van Canneyt
Custom App DB Form	Custom App Framework Database Aware form	Michael Van Canneyt
Custom App Form	Custom App framework form	Michael Van Canneyt

3 Extending `TCustomForm`

When starting from `TCustomForm` such as in `TAppCustomForm`, there are 2 properties which *must* be published before the created form can be handled in the IDE. The IDE expects to be able to set a 'Caption' and 'PixelsPerInch' property. Both these properties exist in `TCustomForm`, but must be explicitly made published - Failing to do so will result in Delphi giving an 'unknown property error' each time the `TCustomForm` descendent is created.

Besides these 2 basic properties, it is now possible to add any possible new (publishable) property and any needed method to the `TAppCustomForm` class. In this article, basically two series of properties will be handled:

1. One series is present in all forms, and concerns printing and saving of form properties, plus navigation to other forms (as introduced in the previous article in this series). This functionality will be implemented in `TAppCustomForm`.
2. The other series of properties and methods is only needed for forms that will get datasets dropped on them. These properties and methods are mainly concerned with the handling of datasets and fields, and will be implemented in a `TAppCustomForm` descendent called `TDBAppCustomForm`. Some of these methods will override the functionality introduced in `TAppCustomForm`.

4 Implementing support for printing

The implementation of support for printing is based on the assumption that information is printed from the place where it is edited/maintained or viewed or in closely related forms, e.g., a list of addresses from a company is printed from the form where these addresses are entered or from the form with the general data of the company, or from the overview of companies. From this assumption, two things follow:

1. One print-out (report) can be asked in multiple forms.
2. The reverse is also true: A form can have multiple print-outs associated with it.

For this reason, with each form, a list of reports is associated. They can, for instance, be stored in a stringlist. (A collection could be used as well). The property which represents this list of reports (introduced in `TAppCustomForm`) will be called `Reports`.

A second property is introduced which is called `DefaultReport`. It contains the name of the report which is printed by default. In a screen with an overview of customers, two reports could be defined: A address list of the customers, or all the available data for the currently selected customer. The `DefaultReport` property determines which of the two reports is printed by default.

To make this work at runtime, the main application form (The MDI form) contains a print menu and associated toolbutton. When this button is pressed (or the menu is activated) the list of associated reports is scanned, and the user is asked which report actually should be printed. Or, the toolbutton could have a style 'tbsDropDown', in which case a popup menu can be associated to it, which contains the names of the reports which are associated to the currently active form. The user can then select at once which print-out he or she wants. Or, if the button is pressed without choosing an item, the default report (obtained from the `DefaultReport` property) is printed.

In this article, the implementation using stringlists is presented: Each string in the stringlist contains a `Name=Value` pair, where the `Name` part is what is shown to the user, the `Value` part contains the actual filename of the report that is to be printed.

The relevant code in the declaration of `TAppCustomForm` can look as follows:

```
TAppCustomForm = Class (TCustomForm)
private
  FReports : TStrings;
  FDefaultReport : String;

Published
  property Reports : TStrings Read FReports Write SetReports;
  property DefaultReport : String Read FDefaultReport
                                  Write FDefaultReport;
```

In the `Create` constructor of `TAppCustomForm`, the stringlist is created, and it is freed again in the destructor `Destroy`.

The strings property can be edited with the standard strings property editor of Delphi, but a customized property editor can also be made. For The `DefaultReport` property, a second customized property editor can be made as well. How to do this will be shown in a later article.

Now that the design time properties are ready, we must implement the actual printing. Until now, nothing has been said about the format of the reports or print-outs. For our discussion, this is not really relevant, as the printing system should determine this.

However, we will assume that the print-out or report is customizable by means of a series of parameters. These parameters can be anything.

- They could describe the type of document: A Quickreport report, a FastReport report, a Word document, an excel sheet, a text file - anything printable.

- They could describe the contents to be printed. In the case of a quitably parametrized quickreport, this could be a parameter that should be inserted in a query; e.g. to print an invoice, the quickreport's underlying dataset could be parametrized so it accepts an invoice number as a parameter. The form should then pass the invoice number of the currently displayed invoice to the printing engine, which can feed it to the quickreport's dataset. For a list of pupils in a school class, the name of the class should be passed on.

For what the parameter is used and how it is implemented in the printing system, will not be treated here, the important idea is that the print-out or report can be parametrized - and most often it will be parametrized. Generally speaking, in a database application, the parameter will be the value(s) of the unique key field(s) of the record to be printed.

In the application framework developed by the author, printing is implemented using Quickreport reports. The designs of the reports are stored in files on disk, and the printing engine reads the report from disk, looks at a query it should execute, fills in any needed parameters, opens the query and then prints the report. The user has an end-user report designer which allows him or her to design her own reports and store them on disk. Of course, the program is delivered with a series of standard reports - namely the reports of which the names are stored in the form.

This means that when the user asks for a printout the following sequence of actions should be carried out:

1. Find out whether the application is ready to print (e.g. save data, prepare some temporary data)
2. Find out whether maybe another report should be printed - depending on the state of the form or the record which is currently active.
3. Collect any parameters for the print. It will be assumed that these parameters can be stored in a stringlist, as a set of `Name=Value` pairs. Here again, another mechanism could be used, e.g., using a collection.
4. Feed the report filename and any parameters to the printing routine.
5. Possibly clean up (delete temporary data etc.) after printing.

The collecting of parameters for a print essentially exists of two parts:

1. A set of standard parameters for this form. These can be stored in a design-time property of type `TStrings`, they will be called 'ReportParams'.
2. A event handler which is called before the printing starts. This handler gets passed the list of parameters, and can modifiy this list by adding or modifying values in the list.

This scheme can be implemented using a number of procedures and events.

The main entry point is the `Print` function, which is called from the `OnClick` handler from the print toolbutton or menu entry.

```
procedure TAppCustomForm.Print(ReportName: String);

Var
  Params : TStrings;
  FileName : String;
```

```
begin
  If (Not DoBeforePrint(ReportName)) or (ReportName='') then
    exit;
  If FReports.IndexOfName(ReportName)=-1 then
    Raise Exception.CreateFmt(SErrNoSuchReport,[ReportName]);
  Params:=TStringList.Create;
  Try
    Params.Assign(ReportParams);
    DoGetPrintParams(ReportName,Params);
    FileName:=DoGetPrintFileName(ReportName);
    DoPrint(FileName,Params);
    DoafterPrint(ReportName,Params);
  finally
    Params.Free;
  end;
end;
```

The first thing that is done is the calling of `DoBeforePrint`; This function will set up anything needed before printing can start - it can also possibly change the name of the report that will be printed. Then a stringlist is created which will contain the parameters for the print. After the list is created, the form's default parameters (reportParams) are copied into it. After that is done, `DoGetPrintParams` is called. This virtual method calls an event handler which allows the programmer to add or modify the list of parameters. (e.g. by asking the user a question, and, based on the answer, insert some extra parameters.)

After the parameters are fetched, an extra handler is called which allows the programmer to change the name of the file that will be used. This is done in the `DoGetPrintFileName` method which will be discussed below. However, By default, this method will return the filename stored in the `Reports` property of the form.

After this the file is actually printed. This is implemented in the `DoPrint` method.

When the `DoPrint` method returns, the `DoAfterPrint` method is called to do any cleaning up, if necessary.

The `DoBeforePrint` function which is used to decide whether a print is possible. By default it calls a `BeforePrint` event handler. It looks as follows:

```
type
  TBeforePrintEvent = Procedure (Sender : TObject;
                                 Var ReportName : String;
                                 Var AllowPrint : Boolean) Of Object;

function TAppCustomForm.DoBeforePrint(var ReportName: String): Boolean;
begin
  result:=True;
  If Assigned(FBeforePrint) then
    FBeforePrint(Self,ReportName,Result);
  If (ReportName='') then
    Result:=False;
end;
```

This function simply calls the `BeforePrint` handler (of type `TBeforePrintEvent`) and checks the result.

The `DoGetPrintParams` procedure can be used to add values to the parameters for the report. By default, it calls an `OnGetPrintParams` event handler of type `TGetPrintParamsEvent`:

```
Type
  TGetPrintParamsEvent = Procedure (Sender : TObject;
                                    ReportName : String;
                                    Params : TStrings) of Object;

procedure TAppCustomForm.DoGetPrintParams(ReportName: String;
  Params: TStrings);
begin
  If Assigned(FOnGetPrintParams) then
    FOngetPrintParams(Self,ReportName,Params);
end;
```

The `DoGetPrintFileName` method is used to determine the filename of the report. By default it returns the filename stored in the `Reports` property, but an `OnGetPrintFileName` event handler can be installed to change this filename.

```
Type
  TGetReportFileNameEvent =
    Procedure (ReportName : String;
               Var ReportFileName : String) Of Object;


function TAppCustomForm.DoGetPrintFileName(ReportName: String): String;
begin
  Result:=Reports.Values[ReportName];
  If Assigned(FOnGetPrinFileName) then
    FOnGetPrintFileName(Self,Result);
end;
```

This can be used to have a single entry in the list of reports, but one which is associated with a series of reports - which report will be printed can depend on the form state or some values of fields in the current record.

- One way in which this is used is when the name of the file to be printed resides in a field of the current record. A single 'Print' entry in the `Reports` property of the form. When this is printed, the name of the form to be printed is retrieved from the current record in the `GetPrintFileName` event handler.
- Another possibility is when the report to be printed is created on-the-fly in some temporary file, and the name of the femporary file is passed on in the `GetPrintFileName` event handler.

After the report has been printed, any clean-up can be done in the `AfterPrint` event handler (of type `TAfterPrintEvent`), which is called in the `DoAfterPrint` method:

```
Type
  TAfterPrintEvent = Procedure (Sender : TObject;
                                ReportName : String;
                                Params : TStrings) Of Object;

function TAppCustomForm.DoAfterPrint(ReportName: String;
                                     Params : TStrings): Boolean;
begin
  If Assigned(FAfterPrint) then
```

```
    FAfterPrint(Self,ReportName,Params);
end;
```

This can be used to remove some temporary data from the database, or delete a temporary file which was created.

The `DoPrint` method does the actual print. Its implementation depends on the actual printing method which is used.

```
procedure TAppCustomForm.DoPrint(ReportName: String; Params: TStrings);
begin
  // Actually print
end;
```

It is not hard to make a small screen which allows the user to add his own report or print-out definitions to the list of reports, or to change the value of the `DefaultReport` property. This screen must simply present the `Reports` property of the form to the user and allow him or her to add definitions to the list. The same form which is used in the design-time property editor for the `Reports` property could be used for this. Also the 'DefaultReport' property should be changeable by the user.

5 Saving the form state

When users close a screen, and open it again later on, they expect some of the screen settings to be saved on exit and restored again when the form is re-opened:

1. The size and position of the form.
2. If there is a database grid: The size and position of the various columns in the grid.
3. If the user can determine the sort order in a grid: What the last used sort order was.
4. If the user changed the 'DefaultReport' property of the form: The new value.
5. If the user added some own report definitions to the list of known reports, these should be saved as well.
6. For a form where data is edited: Default values for the various fields.
7. The user can be allowed to determine which control should get focus when a new record is entered in a dataset. This setting can be saved as well.
8. For faster navigation, the users are allowed to set which data controls have their tab-stop property set to 'True'. This information also is saved.
9. In most forms where an overview of data is presented in a dataset-aware grid, the user is allowed to add any related information to the grid: Any field from a related table can be added to the dataset and is shown in the grid. The list of fields that are added this way is stored as well.
10. If the user can define some filters on data shown in a grid: the definition of the filters should be saved, and the last active filter as well.

This information can be stored for any form in the application by some global routines in the parent form. Sometimes a form needs to save form-specific settings (the 'Checked' property of some checkbox or so); For this, 2 event handlers can be implemented: One to save the settings (when the form is closed), one to restore them (when the form is re-opened).

In addition to that, some application-wide data are also saved on exit and restored when the application is restarted. This information is stored along with the main (MDI) form of the application.

To have this functionality present in all forms of the application, it is obvious that it should be implemented in `TAppCustomForm`. There exist some components, e.g. `TFormStorage` in RXLib or `TOvcFormState` in TurboPower's Orpheus component set, and probably others as well. A solution would be to drop such a component on `TAppForm` and to add all properties that should be saved. However, it can be done differently.

For allow the programmer to customize the saving of settings, a pair of `OnSaveSettings` and `OnRestoreSettings` event handlers are added to the `TAppCustomForm` implementation. This will allow the programmer to save form-specific data as well.

Restoring the form state and saving it should be done when the form is shown and when it is closed, respectively. To do this, the `DoClose` and `DoShow` methods of TCustomForm can be overridden:

Saving the settings is done in the `DoClose` method:

```
procedure TAppCustomForm.DoClose(var Action: TCloseAction);
begin
  inherited;
  If (TAction=caFree) and not (csDesigning in ComponentState) then
    SaveFormSettings;
end;
```

The `Inherited` call will execute the `OnClose` handler of the form. Only when this call returns `caFree` in the Action parameter, i.e. when the form will actually be closed and destroyed, should the settings be saved.

The restoring of the form settings happens in the `DoShow` method of `TAppCustomForm`:

```
procedure TAppCustomForm.DoShow;
begin
  if not (csDesigning in ComponentState) then
    RestoreFormSettings;
  Inherited;
end;
```

The actuall saving and restoring of the settings can happen in many ways. In the application framework developed by the author, a `TFormStorage` component from RXLib is used; The main reason for this is that the application uses the RXLib components, and specifically the `TRXDBgrid` component, which has built-in support for saving the grid state through a `TFormStorage` component. However, the same can be accomplished by using a simple `TIniFile` or `TRegistry` component. In this article an implementation using `TCustomIniFile` is presented, but the same can be done using `TRegistry`.

When saving form settings, two things must be done:

1. Decide whether the form settings must be saved at all. This is mainly for optimization issues: The settings could be stored in a database. In that case, the saving of the settings will take time, and should be avoided when possible. Another reason for this step is that there may be some user preferences that control the saving of settings: Don't save settings at all, save some settings, or save all settings.

2. if step 1 determined that the settings must be saved: Actually save the settings.

The whole should be in a `Try..Except` block - if the saving of settings fails and raises an exception, the form will not close, and it will not be possible to close the form at all since any attempt to close the form again will fail.

This can be implemented as follows:

```
procedure TAppCustomForm.SaveFormSettings;

Var
  F : TCustomInifile;

begin
  Try
    If NeedSaveSettings then
      begin
      F:=TIniFile.Create(Format('%s\%s.ini',[SettingsPath,ClassName]);
      try
        DoSaveSettings(F);
      finally
        F.Free;
      end
  except
    On E : Exception do
      MessageDlg(Format(SErrSavingSettings,[E.Message]),mtWarning,mkOK,0);
  end;
end;
```

After determining whether the settings must be saved, a `TiniFile` object is created. In the above approach, a .ini file per form will be created in the directory determined by the `SettingsPath` variable. The name of the file is taken to be the classname of the form - if multiple instances of the same form are created and shown at the same time, they will write to the same file, which would not be the case if for instance the `Name` property of the form were chosen. Note that another approach using a single .ini file could be used, or the registry could be used.

The `NeedSaveSettings` function determines whether any settings need saving. This is definitely the case if the programmer has attached a `OnSaveSettings` handler, and could be further controlled by a global boolean variable `SaveSettingsOnExit`:

```
function TAppCustomForm.NeedSaveSettings : Boolean;

begin
  Result:=SaveSettingsOnExit or Assigned(FOnSaveSettings);
end;
```

This function is virtual so it can be overridden in descendent form classes.

If the `NeedSaveSettings` function returns true, then the `DoSaveSettings` procedure is executed:

```
Const
  Section = 'PositionSize';

procedure TAppCustomForm.DoSaveSettings(Ini : TCustomIniFile);
```

```
begin
  With Ini do
    begin
    WriteInteger(Section,'WindowState',ord(WindowState));
    If (WindowState=wsNormal) then
      begin
      WriteInteger(Section,'Top',Top);
      WriteInteger(Section,'Left',Left);
      WriteInteger(Section,'Width',Width);
      WriteInteger(Section,'Height',Height);
      end;
    If ActiveControl<>Nil then
      WriteString(Section,'ActiveControl',ActiveControl.Name)
    else
      WriteString(Section,'ActiveControl','');
    end;
  If Assigned(FOnSaveSettings) then
    FOnSaveSettings(Self,Ini);
end;
```

As the last step, the `OnSaveSettings` handler, if one is assigned, is executed. It is of type `TSettingsEvent`:

```
Type
  TSettingsEvent = Procedure (Sender : TObject;
                              Ini : TCustomIniFile) of Object;
```

The `Ini` parameter can be used by the programmer to save any settings he sees fit. Obviously, the section with the position and size of the form should not be touched by the `OnSaveSettings` handler.

Restoring the form's state is simply the reverse action of saving the settings:

```
procedure TAppCustomForm.RestoreFormSettings;

var
  F : TCustomIniFile;

begin
  try
    If NeedRestoreSettings then
      begin
      F:=TIniFile.Create(Format('%s%s%s.ini',[SettingsPath,
                                              PathDelimiter,
                                              ClassName]);
      try
        DoRestoreSettings(F);
      finally
        F.Free;
      end
  except
    On E : Exception do
      MessageDlg(Format(SErrRestoringSettings,[E.Message]),
                 mtWarning,mkOK,0);
  end;
end;
```

Needless to say that the state-restoring process should open the same .ini file as the state-saving process.

The `NeedRestoreSettings` is similar to the `NeedSaveSettings` function:

```
function TAppCustomForm.NeedRestoreSettings : Boolean;

begin
  Result:=SaveSettingsOnExit or Assigned(FOnRestoreSettings);
end;
```

The reason for implementing this check in 2 separate functions is that the `NeedSaveSettings` function could be optimized to return `False` when no form settings have changed after they were restored. In case the settings are stored in a database, this will save valuable time.

The actual restoring of the settings is straightforward:

```
procedure TAppCustomForm.DoRestoreSettings(Ini : TCustomIniFile);

Const
  Section = 'PositionSize';
  I : Integer;
  S : String;
  C : TWinControl;

begin
  With Ini do
    begin
    I:=ReadInteger(Section,'WindowState',Ord(WindowState));
    WindowState:=TWindowState(I);
    If (WindowState=wsNormal) then
      begin
      Top:=ReadInteger(Section,'Top',Top);
      Left:=ReadInteger(Section,'Left',Left);
      Width:=ReadInteger(Section,'Width',Width);
      Height:=ReadInteger(Section,'Height',Height);
      end;
    S:=ReadString(Section,'ActiveControl','');
    If (S<>'') then
      begin
      C:=TWinControl(FindComponent(S));
      If (C<>Nil) then
        ActiveControl:=C;
      end;
    end;
  If Assigned(FOnRestoreSettings) then
    FOnRestoreSettings(Self,Ini);
end;
```

Note that the restoring process does not destroy any design-time settings in case the .ini file happens to be empty.

Descendent screens can now override the `NeedRestoreSettings` and the `NeedSaveSettings` functions as well as the `DoRestoreSettings` and `DoSaveSettings` to save and restore additional settings. For instance for a DB-aware descendent of `TAppCustomForm` this will be used to store several settings.

Note that more settings can be saved and restored than have been shown here; Which properties are saved depends entirely on the implementation of `TAppCustomForm`. For instance the value of the `DefaultReport` property introduced earlier could be saved. If the user is allowed to add report definitions to the `reports` property, then these could be saved as well.

6 Implementing a DB-Aware form.

The functionality introduced so far is not concerned with datasets; Not all forms in an application are concerned with datasets; Those that do, however, also can have some functionality in common, and this functionality can be implemented in a separate `TAppCustomForm` descendent.

What functionality can be introduced in a DB-aware form ? There are basically 2 kinds of functionality which can be introduced:

1. Visual functionality: DB-Aware screens can introduce a uniform look: A navigator bar at a fixed location, a status bar showing the current state of the dataset, an edit control which can be used for looking up records.
2. Non-visual functionality: When closing the form, the form should check whether there is any unsaved data (i.e. a dataset which is still in edit or insert mode) and take appropriate action if there is such data. It could save and restore additional information about the datasets. For forms that link to other forms, functionality can be introduced that establishes master-detail links between forms.

This article will be concerned mostly with the latter kind, as the `TAppCustomForm` descendent is not visual, it will introduce only non-visual functionality. The following functionality will be introduced:

- Opening of datasets that are present on the form when the form is opened. This is especially important in applications where multiple screens work on a single database connection.
- When the form is closed, the form will look for unsaved data and will deal with it in the appropriate way.
- The printing process will be enhanced: Field values can be passed as parameters to the printing system.
- The form saving/restoring process will be enhanced: Some additional properties will be saved.

To illustrate the ideas, a `TAppDataset` descendent of `TClientDataset`, `TQuery` or `TADOQuery` is implemented which has some additional properties:

AutoOpen This is a boolean property which determines whether the dataset will be opened automatically at runtime or not.

DataSource This (dummy) property is introduced only in a `TClientDataset` descendent, as it is standard implemented in `TQuery` or `TDAOQuery`.

DefaultValues This is a series of default values that are stored in the fields when the dataset goes to insert mode. This overlaps a bit with the `DefaultExpression` property of TField, but the disadvantage of that property is that it is not observed in all `TDataset` descendents. The implementation here works always, and happens at the time of insert, i.e. the user is aware of them as soon as the insert happens. This is in contrast with the Default values introduced in a database SQL server, which are only applied after the record is posted in the database.

FocusControl This is a property of type `TWinControl`. If it is set, the control indicated will get focus when the dataset goes to insert mode. This way the user starts the input of a new record always with the same field. The user is permitted to change the value of this property at runtime.

Furthermore, `TAppDataset` introduces 2 methods `SaveSettings` and `RestoreSettings`, which save the `DefaultValues` and `FocusControl` setting.

More functionality can of course be introduced, but to illustrate the ideas, the above will be sufficient. The `TDBAppCustomForm` knows about the `TAppDataset` properties and is able to handle them. For instance, it could be made to respond to a keystroke combination which would save all current field values as default values, or another combination which would only do this for the field whose value is edited in the control which currently has focus.

7 Passing field values to the printing system

The DB aware form introduces functionality which interacts with the printing system. More specifically, it gives design-time functionality to pass field values to the printing system as report parameters. To be able to do this, a way is needed to enumerate report parameters and the fields from which they should be filled. This will be done in a collection, `TFieldToReportParams`, which has items of class `TFieldToReportParamsItem`. This class is defined as follows:

```
TFieldToReportParamsItem = Class(TCollectionItem)
private
  FDataField: String;
  FReportParam: String;
  FDataSource: TDataSource;
Published
  Property DataSource : TDataSource Read FDataSource Write FDataSource;
  Property DataField : String Read FDataField Write FDataField;
  Property ReportParam : String Read FReportParam Write FReportParam;
end;
```

The meaning of these properties should be clear: When the parameters for the print-out are collected, each of the items in the collection will be examined, and the value of the `DataField` field in the dataset of `Datasource` will be added to the report parameters with name `ReportParam`. By adding items to the collection and setting the appropriate properties, the programmer can decide what field values get passed on to the printing system as parameters.

The collection is a simple `TCollection` descendent which contains a reference to the form that maintains it.

```
TFieldToReportParams = Class(TCollection)
```

```
Private
  FForm : TDBAppCustomForm;
end;
```

The printing system collects parameters through the `DoGetPrintParams` method. This method is overridden in `TDBAppCustomForm` and is a simple loop:

```
procedure TDBAppCustomForm.DoGetPrintParams(ReportName: String;
                                            Params: TStrings);

Var
  I : Integer;
  S : String;

begin
  Inherited;
  For I:=1 to FFieldToReportParams.Count-1 do
    With FFieldToReportParams.Items[i] as TFieldToReportParamsItem do
      begin
      S:=DataSource.Dataset.FieldByName(DataField).asString;
      Params.Values[ReportParam]:=S;
      end;
end;
```

To make sure that the data is present in the database when printing, a boolean `PostOnPrint` property is introduced in `TDBAppCustomForm`. If this property is set to `True`, then the form will save all datasets before printing.

This is done by overriding the `DoBeforePrint` method:

```
function TDBAppCustomForm.DoBeforePrint(var ReportName: String): Boolean;
begin
  Result:= Inherited DoBeforePrint(reportName);
  If (Result and PostOnPrint) then
    PostDatasets(False);
end;
```

The `PostDatasets` method will be discussed later. If an error occurs during the posting of the datasets an exception will be raised and no printing will occur.

8 Opening and closing all datasets

The opening and correct closing of datasets is also something that is taken care of by the form. In a database application, the location of the database or the remote server (for MIDAS Client datasets) is often variable, and should be set before the datasets are opened. This means that the Datasets on the form should be closed when the form is designed. If they were active when the form was last saved in the IDE, the form loading mechanism will attempt to open them again. If the database object is not yet properly set up, this will lead to problems.

Therefore, the form should do whatever is necessary to initialize the database connection, and only when this is done, all datasets should be opened. To minimize loading time, datasets that are in a master-detail relationship should be opened in the correct order; If not, there will be unnecessary reloading of data: If dataset A is opened first, and it needs parameters from Dataset B (for instance a parameter with name CustID) it attempts to retrieve these. If Dataset B is still closed, then the value of the parameter will be `Null`. When Dataset B is opened afterwards, it will notify Dataset A of this fact, causing dataset A to reload itself, this time with an existing value for the parameter `CustID`. If Dataset B is opened first, then Dataset A will be loaded only once, since the correct value for the CustID parameter will be retrieved when dataset A is opened.

A second aspect to consider is that maybe not all datasets should be opened when the form is opened, e.g. a query used to look up special values. To be able to customize this, a boolean property `AutoOpen` is introduced in `TAppDataset` which controls the automatic opening of the dataset.

To enable the programmer to take specific actions before and after all datasets have been opened, two event handlers are defined: `BeforeOpenDatasets` and `AfterOpenDatasets`. The former could be used to set up extra parameters in the datasets, the other can be used to set up various display properties in the form such as displaying record counts etc.

With all needed properties in place, the `DoShow` method can be overridden in `TDBAppCustomForm`:

```
procedure TDBAppCustomForm.DoShow;
begin
  inherited;
  OpenDatasets;
end;
```

The `OpenDatasets` method is a simple loop and looks as follows:

```
procedure TDBAppCustomForm.OpenDatasets;

Var
  I : Integer;

begin
  DoBeforeOpenDatasets;
  For I:=0 to ComponentCount-1 do
    If Components[i] is TAppDataset then
      OpenDataset(Components[i] as TAppDataset);
  DoAfterOpenDatasets;
end;
```

The `DoBeforeOpenDatasets` and `DoAfterOpenDatasets` simply call the corresponding event handlers if they are installed:

```
procedure TDBAppCustomForm.DoBeforeOpenDatasets;

begin
  If Assigned(FBeforeOpenDataSets) then
    FBeforeOpenDataSets(Self);
end;

procedure TDBAppCustomForm.DoAfterOpenDatasets;
begin
  If Assigned(FAfterOpenDataSets) then
    FAfterOpenDataSets(Self);
end;
```

The `OpenDataset` method contains the needed logic to open a dataset, respecting any master-detail relations that might be present:

```
procedure TDBAppCustomForm.OpenDataset(DataSet : TAppDataSet);

begin
  With Dataset do
     If Not Active and AutoOpen then
       begin
       If Assigned(DataSource) and
          Assigned(Datasource.Dataset) and
          (Datasource.Dataset is TAppDataset) then
          OpenDataset(Datasource.Dataset as TAppDataset);
       Open;
       end;
end;
```

As can be seen, it recursively calls itself if it needs to open a master dataset in a master-detail relationship.

When closing a form with datasets where the user can edit data, a check should be done to see whether all data is saved, and if so, close all datasets. If not, appropriate action should be taken:

1. Cancel all edits automatically.
2. Post all edits automatically.
3. Ask the user what should be done.

The choice between these three options can be left to the user by introducing a global variable `ModifiedAndCloseAction`:

```
type
  TModifiedAndCloseAction = (macPost,macCancel,macConfirm);

Var
  ModifiedAndCloseAction : TModifiedAndCloseAction = macConfirm;
```

It is not hard to allow the user to set the value of this variable in a globale preferences dialog. The programmer can override this value through a boolean property `PostOnClose` which, if set to `True`, will force all datasets to be posted if the form is closed. This may sometimes be necessary to force integrity of the database.

The check for unsaved data can be implemented in the `CloseQuery` function. This function is implemented in `TCustomForm` and can be overridden. It should return `True` if the form can be closed, and `False` if not;

```
function TDBAppCustomForm.CloseQuery: Boolean;
begin
  Result:=Inherited CloseQuery;
  If Result then
    CloseQuery:=CloseAllDatasets;
end;
```

The call to inherited will make sure the `OnCloseQuery` handler installed by the programmer is called. If this returns `True`, then the saved-data check is done. If the programmers handler returns `False`, the check is omitted.

The `CloseAllDatasets` returns `true` if all datasets were already posted or have been posted succesfully, and returns `False` if the posting of one of the datasets failed. It works as follows:

```
function TDBAppCustomForm.CloseAllDatasets: Boolean;

Var
  I : integer;

begin
  Result:=Not DatasetsInEdit;
  If (Not Result) then
    if PostOnClose then
      result := PostDatasets(False)
    else
      Case ModifiedAndCloseAction of
        macCancel  : Result:=Canceldatasets;
        MacPost    : Result:=PostDatasets(False);
        MacConfirm : case MessageDLG(SSaveData,mtInformation,mbYesNoCancel,0) of
                       mrNo :  Result := CancelDatasets;
                       mryes : result := PostDatasets(True);
                     else
                       result := false;
                     end;
      end;
  If Result then
    For I:=0 to ComponentCount do
      If Components[i] is TDataset then
        (Components[i] as TDataset).Close;
end;
```

The first thing that the `CloseAllDatasets` method does, is check whether there are still datasets in edit mode; If so, then all datasets on the form are posted or canceled, or the user is queried for the needed action.

The `DatasetsInEdit` function is a simple loop which checks whether a dataset is in edit mode (i.e. its state is `dsEdit` or `dsInsert`):

```
function TDBAppCustomForm.DatasetsInedit: Boolean;

var
  I : Integer;

begin
  I:=0;
  Result:=False;
  While (I>=0) and Not Result do
    begin
    If Components[i] is TDataset then
      Result:=TDataset(Components[i]).State in dsEditModes;
    Dec(i);
    end;
end;
```

Canceling of edits is also implemented in a simple loop:

```
function TDBAppCustomForm.CancelDatasets: Boolean;

var
  I : Integer;

begin
  For I:=0 to ComponentCount-1 do
    If Components[i] is TDataset then
      With TDataset(Components[i]) do
        If State in dsEditModes then
          Cancel;
  Result:=True;
end;
```

The actual posting of the datasets is also a loop. As a parameter it accepts a boolean which queries the user whether the modified data should be saved. This parameter will be set to `False` if the user already was queried, and will be set to `True` if the posting is automatically executed:

```
function TDBAppCustomForm.PostDatasets(NoConfirmation : Boolean): Boolean;

Var
  I : Integer;

begin
  Result:=True;
  I:=ComponentCount-1;
  While Result and (I>=0) do
    begin
    If Components[i] is TDataset then
      Result:=DoPost(TDataset(Components[i]) ,NoConfirmation);
    Dec(I);
    end;
end;
```

The `DoPost` does the actual posting of the data. It returns `True` if the dataset was posted, false if not.

```
function TDBAppCustomForm.DoPost(DS : TDataset; NoConfirmation : Boolean) : Boolean;

begin
  Result:=Not (DS.State in [dsEdit,dsInsert]);
  If Not Result then
     If NoConfirmation or ConfirmPost Then
       begin
       DS.Post;
       Result:=True;
       end;
end;
```

The `ConfirmPost` function simply asks the user whether the data should be posted and returns `True` if the dataset should be posted:

```
function TDBAppCustomForm.ConfirmPost : Boolean;
```

```
begin
  Result:=MessageDLG(SSaveData,mtInformation,mbYesNoCancel,0)=mrYes;
end;
```

9 Saving additional form state data

Another functionality of the DB-aware form is that some properties of `TAppDataset` are also saved and restored when the form is closed and opened, respectively.

To do this, the DB-Aware form overrides the `DoRestoreSettings` and `DoSaveSettings` methods:

```
procedure TDBAppCustomForm.DoRestoreSettings(Ini: TCustomInifile);

Var
  I : Integer;

begin
  inherited;
  For I:=0 to ComponentCount-1 do
    If Components[i] is TAppDataset then
      TAppdataset(Components[i]).RestoreSettings(Ini);
end;

procedure TDBAppCustomForm.DoSaveSettings(Ini: TCustomIniFile);

Var
  I : Integer;

begin
  inherited;
  For I:=0 to ComponentCount-1 do
    If Components[i] is TAppDataset then
      TAppdataset(Components[i]).SaveSettings(Ini);
end;
```

As can be seen, the actual saving of the dataset properties is delegated to the `TAppDataset` class.

To make sure that these methods are called, the form should override the `NeedSaveSettings` and `NeedRestoreSettings`:

```
function TDBAppCustomForm.NeedRestoreSettings: Boolean;
begin
  Result:=(Inherited NeedRestoreSettings) or HasAppDatasets;
end;

function TDBAppCustomForm.NeedSaveSettings: Boolean;
begin
  Result:=(Inherited NeedSaveSettings) or HasAppDatasets;
end;
```

The `HasAppDatasets` simply checks whether any `TAppDataset` component was dropped on the form. This check could be refined by checking whether the `TAppDataset` component actually needs to save or restore a setting.

```
function TDBAppCustomForm.HasAppDatasets: Boolean;

Var
  I : Integer;

begin
  I:=ComponentCount-1;
  Result:=False;
  While not result and (I>=1) do
    begin
    Result:=Components[i] is TAppDataset;
    Dec(i);
    end;
end;
```

The `SaveSettings` and `RestoreSettings` methods of the `TAppDataset` component do the actual work of saving and restoring settings. They try to do this as optimal as possible:

```
procedure TAppDataset.RestoreSettings(Ini: TCustomInifile);

Var
  I : Integer;
  S, Section : String;

begin
  With Ini do
    begin
    If Not ReadOnly then
      begin
      Section:=Name+SDefaults;
      FDefaultValues.Clear;
      I:=ReadInteger(Section,'Count',-1);
      While (I>=0) do
        begin
        S:=ReadString(Section,Format('Field%d',[i]),'');
        If S<>'' then
          FDefaultValues.add(StringReplace(S,',','=',[]));
        Dec(i);
        end;
      end;
    Section:=Name+SPreferences;
    S:=ReadString(Section,'FocusControl','');
    If (S<>'') then
      FocusControl:=Owner.FindComponent(S) as TWinControl;
    end;
end;
```

Note that each `TAppDataset` component retrieves and stores its settings in two private sections of the `.ini` file which are composed of the name of the component. This ensures that the settings of 2 datasets will not overwrite each other.

The saving of the settings is the opposite process:

```
procedure TAppDataset.SaveSettings(Ini: TCustomInifile);

Var
  I : Integer;
  S,Section : String;

begin
  With Ini do
    begin
    If Not ReadOnly then
      begin
      Section:=Name+SDefaults;
      EraseSection(Section);
      WriteInteger(Section,'Count',FDefaultValues.Count);
      for I:=0 to FDefaultValues.Count-1 do
        begin
        S:=StringReplace(FDefaultValues[i],'=',',',[]);
        WriteString(Section,Format('Field%d',[i]),S);
        end;
      end;
    Section:=Name+SPreferences;
    If Assigned(FocusControl) then
      WriteString(Section,'FocusControl',FocusControl.Name)
    else
      DeleteKey(Section,'FocusControl');
    end;
end;
```

The actual use of these `TAppDataset` properties will be discussed in a later article.

Note that, as argued earlier, more properties could be saved, for example, the sizes and positions of columns in a TDBGrid, or the key of the current record, so the dataset can be repositioned to this record when it is opened again. The implementation of such things is quite straightforward and will not be presented here.

10 Conclusion

In this article an attempt was made to show how to add some common functionality to forms that are part of a large (database) application. The introduction of this functionality in a parent form ensures that the functionality is present in all forms, without any extra work on behalf of the programmer. Implementing this functionality not in a TForm descendent but in a custom form makes sure that the functionality will never be inadvertently overridden by a missing call to `Inherited` in a descendent `TForm` instance.

This, combined with some wizards, makes the life of the programmer easier: e.g. the check for unsaved data is handled by the form and need not be implemented over and over again. Similarly, saving of settings is reduced to a double click on the form's `OnSaveSettings` and `OnRestoreSettings` event handlers in the Delphi IDE. This approach also puts more power in hands of the user, which can customize the (uniform) look and feel of his application: The saving and restoring of form properties make the user experience more smooth, if enough form properties are saved.

In the next article more ways of letting the user control the application will be presented by making full use of the possibilities of `TClientDataset`.