Building A Database Expert

by Marco Cantu and Bob Swart

How many times have you used the Delphi Database Form Expert? It is really invaluable for starting the development of a data entry form, or of any other form related to one or more database tables. After a while, however, you might want to customize it, to have standard elements in your form by default, such as a company logo or some special capability...

Of course, you could add those capabilities and change the way the data is presented after the form has been generated, but you might end up making the same changes over and over again. It would be better to customize the Database Form Expert to generate the forms as you want them in the first place.

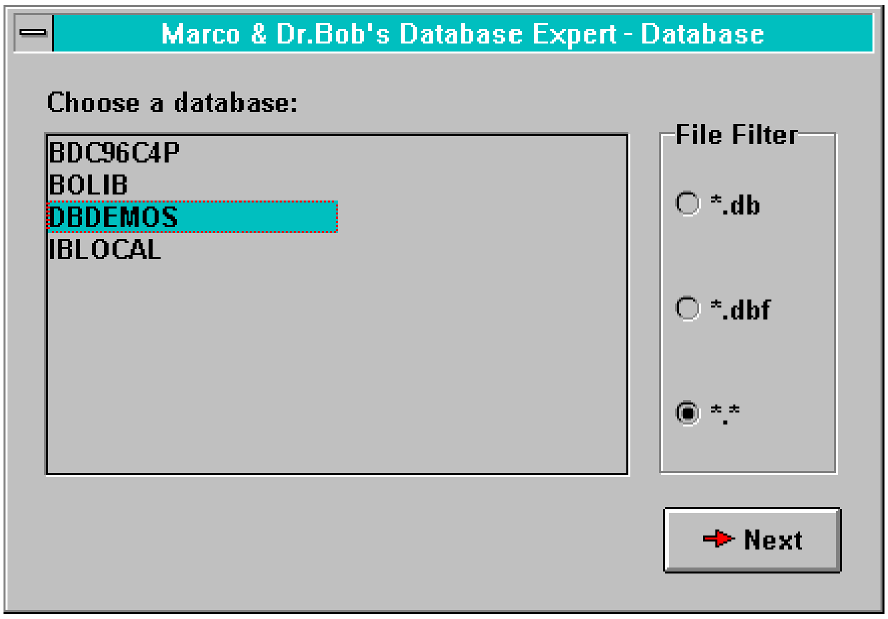
There is only one problem: Delphi includes the source code of other less powerful experts, but no code for the Database Form Expert. So, to customize this tool we need to write it again from scratch! This is exactly the aim of this article. We recently made a presentation together about dynamic database code and Delphi experts and invented this common example.

As the aim of this article is to build a Database Expert, we will delve into some advanced Delphi programming topics: class references, the `Session` component, streaming forms and experts.

An Expert In A Notebook

The first part of the work for the Database Expert is to build an application capable of generating database forms at run-time. The second part will be to turn the program into an expert, capable of generating the source file for a form.

Both examples are based on a form having the same structure: a notebook. Instead of having tabs connected, the notebook uses buttons to move from one page to the next. The button to move between the pages is usually not enabled until the current selection has been completed. For example, you cannot move to the second page (showing a list of tables) until you have selected a database alias in the first page (see Figure 1).


Marco & Dr.Bob's Database Expert - Database

Choose a database:

BDC96C4P
BOLIB
DBDEMOS
IBLOCAL

File Filter
○ *.db
○ *.dbf
● *.*

➤ *Figure 1*

After these first two pages, the user can select some of the fields from the table and then give a caption to each of the labels corresponding to the fields. Another feature of this form is that its caption always displays the name of the current page of the notebook. This is obtained with a simple handler for the `OnPageChanged` event of the notebook, containing the line:

```
Caption := 'DataViewer - ' +
  NoteBook1.ActivePage;
```

Using a notebook is a good way to show different controls on a single form, depending on the current status. Notice that the two buttons used to move back and forth are replicated in each page (although it is possible to place a single button in each page of the form). The reason for this choice is that most of the code of the program is actually executed in the `OnClick` event of the `Next` buttons.

Dynamic Database Programming

At the beginning, the program should let a user choose a table from a database. To be able to explore the existing databases dynamically, we have to use the global `Session` object, of class `TSession`, which is defined in the DB unit, and handles data access sessions. In short the `Session` component holds a list of databases the application is connected with (in the `Database` array property, see also the `DatabaseCount` property), and can be used to set a password as well as alter the current database connections at run-time.

It is a useful component when you want to know BDE configuration information, such as a list of aliases, and can be used to create a list of the tables in a database. Table 1 shows the list of `Get...` methods for the `Session` component. Notice that they all have a `TStrings` parameter which is used to pass the information back to the calling application. For more details see the Delphi help.

The first call to the `Session` component takes place when the form is created:

```
procedure TForm1.FormCreate(
  Sender: TObject);
begin
  Session.GetDatabaseNames(
    DatabaseList.Items);
  Notebook1.PageIndex := 0;
end;
```

We've used the Items stringlist as a parameter of the GetDatabaseNames method to give the destination listbox. As an alternative we could have declared and created a TStringList object, passed it as a parameter to the procedure and then assigned it to the Items property of the listbox. The second line of the method above is a must-have statement for any notebook based form: regardless of the last page you were working on at design time, the notebook will always start up on the first page. Once the user selects an item in the database listbox, the Next button is enabled:

```
procedure TForm1.DatabaseListClick(
 Sender: TObject);
begin
  BitBtnNext1.Enabled := True;
end;
```

This enabling after the selection approach is used in each page, although we won't show it again in the following discussion. The other component of this first page is used to select a filter for the database. The available filters have been prepared with local tables in mind, but you can adapt them to SQL server databases or simply omit them. The filters are the strings of a list associated with a RadioGroup component, so they might even be customizable by the user. As you'll see in a while, the only thing we'll need is the string corresponding to the selected item, so we'll use this RadioGroup exactly as a listbox.

Clicking on the Next button of the first page, now enabled, retrieves the list of tables, again using the Session global object, into a listbox on the second page (see Listing 1).

After retrieving the strings with the current database and filter, GetTableNames is called, again passing the Items of a listbox as the last parameter. The result is in Figure 2. Of course, the page is changed too.

```
procedure GetAliasNames(List: TStrings);
procedure GetDatabaseNames(List: TStrings);
procedure GetDriverNames(List: TStrings);
procedure GetDriverParams(const DriverName: string; List: TStrings);
procedure GetTableNames(const DatabaseName, Pattern: string;
  Extensions, SystemTables: Boolean; List: TStrings);
procedure GetStoredProcNames(const DatabaseName: string;
  List: TStrings);
```

➤ *Table 1: Methods of the Session component you can use to retrieve database information*

```
procedure TForm1.BitBtnNext1Click(Sender: TObject);
var
  CurrentDB, CurrentFilter: string;
begin
  CurrentDB := DatabaseList.Items [DatabaseList.ItemIndex];
  CurrentFilter := FilterGroup.Items[FilterGroup.ItemIndex];
  Session.GetTableNames(CurrentDB, CurrentFilter,
    True, False, TableList.Items);
  NoteBook1.PageIndex := 1;
  BitBtnNext2.Enabled := False;
end;
```

➤ *Listing 1*

The Fields Definition

Using the Session component we have been able to reach a specific database table. Now we need to know what fields are in this table. When you open a table, you can find information about the structure of the table in the Fields array property. However, we do not want to open the table, because this can take some time. The Table component has a less well-known property, FieldDefs, which stores the definition of the fields.

This is an object of type TFieldDefs, in practice an array of TFieldDef structures, with the following properties: FieldNo, Name, DataType, FieldClass, Required and Size. The DataType property holds the original definition in the database, the FieldClass indicates the TFields descendant that will be added to the Fields array when the table is opened. You can access each property of the field definition objects even if the table is not open, but you have to call the Update method of the FieldDefs property.

To use this approach, we've added a Table component to the form, and set its DatabaseName and TableName properties using the selected items from the listboxes on the first two pages of the notebook. Then we've called the Update method of the FieldDefs, as you can see in Listing 2. At this point, we can scan the FieldDefs list and fill a new listbox with the field names and the corresponding class names for the related TField object. You can see the result of this code in Figure 3.

This third page of the notebook allows a user to select fields from the table. Only the selected fields will be present in the form. The listbox has the MultiSelect property set to True and ExtendedSelect to False, so that clicking on an item reverses its current selection status. There are also buttons to select all or none of the fields.

What Is A Class Reference?

Now we have to step aside for a moment and briefly look at what a class reference is. Maybe you already know, but most Delphi programmers are not aware of this feature of the language. A class reference is basically a variable storing a class data type instead of a value. You can define a new class reference type, and a variable of that type, as follows:

```
type CRef = class of TControl;
var  CRef1: CRef;
```

The CRef1 variable can receive as a value the TControl class (the class used in the definition of its type) or any of its subclasses. From that point on you can use the class reference where a data type is expected. You can use it in a class comparison, to get the class name, but also to create a new object of that class:

```
Control1 :=
  CRef1.Create(Form1);
```

This line of code can be used to create an object of any TControl subclass, depending on the class assigned to CRef1. We will do exactly the same to create the data-aware controls in the form, but we'll use class references first to find a match between TField subclasses and the class of a data-aware control. Notice that the VCL already defines many class references, as TClass (related to the generic TObject), TComponentClass, TControlClass and TFieldClass.

From Data Types To Controls

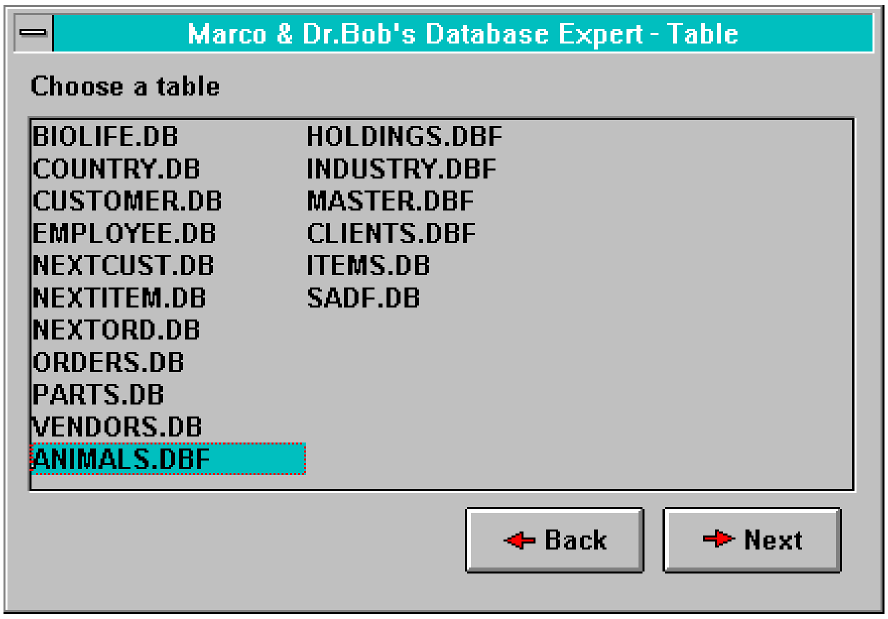
To map the data types of the database fields to data-aware components we've written some code based on class references. The translation function has the following prototype:

```
function ConvertClass(
  FieldClass: TFieldClass) :
  TControlClass;
```

Its code basically scans an array of class references mapping the two types and defined as:

```
type
  CVTable =
    array [1..FieldTypeCount,
    1..2] of TClass;
const
 ConvertTable: CVTable = (...);
```

You can see the full definition of the constant array and the conversion function in Listing 3.

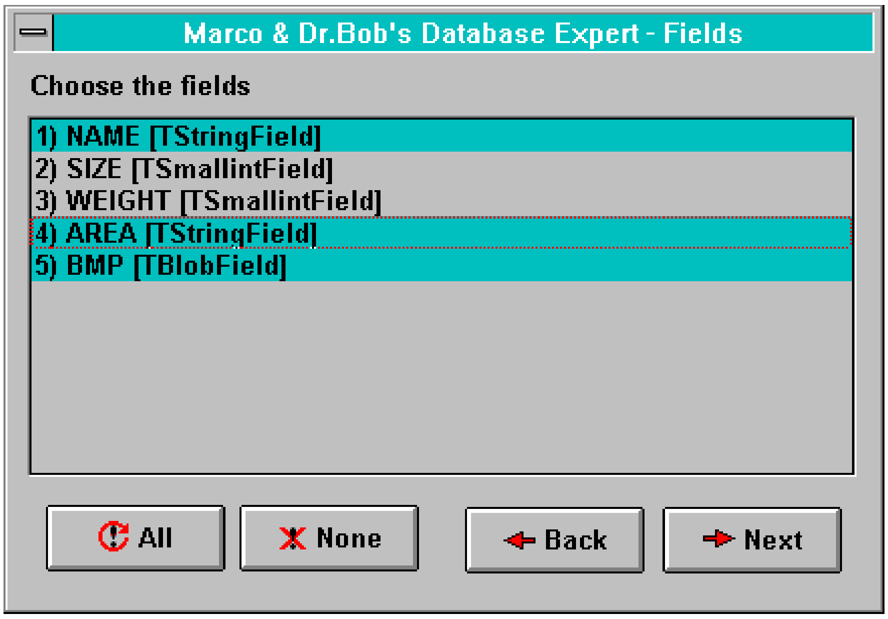

Marco & Dr.Bob's Database Expert - Table

Choose a table

BIOLIFE.DB	HOLDINGS.DBF
COUNTRY.DB	INDUSTRY.DBF
CUSTOMER.DB	MASTER.DBF
EMPLOYEE.DB	CLIENTS.DBF
NEXTCUST.DB	ITEMS.DB
NEXTITEM.DB	SADF.DB
NEXTORD.DB	
ORDERS.DB	
PARTS.DB	
VENDORS.DB	
ANIMALS.DBF	

➤ Above: Figure 2

➤ Below: Figure 3


Marco & Dr.Bob's Database Expert - Fields

Choose the fields

1) NAME [TStringField]
2) SIZE [TSmallintField]
3) WEIGHT [TSmallintField]
4) AREA [TStringField]
5) BMP [TBlobField]

```
procedure TForm1.BitBtnNext2Click(Sender: TObject);
var I: Integer;
begin
  {set the properties of a table}
  with Table1 do begin
    DatabaseName := DatabaseList.Items[DatabaseList.ItemIndex];
    TableName := TableList.Items[TableList.ItemIndex];
    {load the fields definition}
    FieldDefs.Update;
  end;
  {clear the list then fill it}
  FieldList.Clear;
  for I := 0 to Table1.FieldDefs.Count - 1 do
    {add number, name, and class name}
    FieldList.Items.Add (Format('%d) %s [%s]',
      [Table1.FieldDefs[I].FieldNo, Table1.FieldDefs[I].Name,
      Table1.FieldDefs[I].FieldClass.ClassName]));
  NoteBook1.PageIndex := 2;
  BitBtnNext3.Enabled := False;
end;
```

➤ Listing 2

The first thing to notice is that the association, as defined by the array, is far from perfect. In particular, deciding that Blob fields should map to DBImage controls is a guess reasonable for dBase tables only. The second thing to notice is that the conversion function simply scans the table looking for a match in the first column. When the match is found, the value of the second column is used. The function has to make some casts between class references, but the rest is clear enough.

Editing The Labels

The fourth page of the notebook displays the field name and the corresponding control (computed with the above function) of the selected fields in the first column of a string grid (see Figure 4).

The second column displays the field name again, but you can edit it, to provide text for the descriptive label that will be placed near the control. You can see the code used to fill this StringGrid in Listing 4 (taken from the BitBtnNext3Click method).

The for loop in this code uses a second counter (RowNum) for the current row of the StringGrid. In fact, if some fields are selected, the index of the fields will not match with the rows of the grid.

There is some awkward code in this listing, too. In particular, to the result of the ConvertClass function (a class reference) we apply the ClassName class method to obtain a textual description of the class. Notice that you can only apply a class reference to class methods, not standard methods, because no class instance is associated with a class reference.

In this fourth page of the notebook you can customise the labels and check that the fields and their controls match properly. We've provided no way to choose a control class, but this could be a nice addition to the program. Pressing the Generate button makes the new form appear.

Creating The Form

Once you have the class references to the controls, and the captions of the labels, you have only to arrange everything on a form. We've already prepared a standard form, which is part of the program, with a toolbar hosting a DBNavigator and a scrollbox covering the rest of the client area (and aligned to it). The new components will be placed in the scrollbox, so that if there are too many the scrollbar will refer only to that portion of the form, and the toolbar will remain at its place (and not scroll). Incidentally, this is the same approach used by the Delphi Database Form Expert.

The form also has a Table component, which is connected at runtime with the database and table selected by the user. This is the first operation done after the form has been created and is quite simple. The core of this procedure is the creation of labels and controls: see Listing 5.

The code use to create the label is the typical code used to create controls at runtime. Notice that the owner of the control is the form (this is indicated in the parameter of the class constructor, Create), while the Parent of the control is the ScrollBox, because the label needs to go inside it. As a caption of the label the program uses the value inserted by the user. The name of the label is based on the loop counter: its horizontal position is fixed and its vertical position is increased each time, computing the height of the last control and leaving some blank space (this code is not shown in the listing).

The second part of the code in Listing 5 is used to create the control and is based on the class reference returned by the conversion function. The Name of this

➤ *Listing 3*

```
const ConvertTable: CVTable = (
  (TStringField, TDBEdit),
  (TIntegerField, TDBEdit),
  (TSmallintField, TDBEdit),
  (TWordField, TDBEdit),
  (TFloatField, TDBEdit),
  (TCurrencyField, TDBEdit),
  (TBCDField, TDBEdit),
  (TBooleanField, TDBCheckBox),
  (TDateTimeField, TDBEdit),
  (TDateField, TDBEdit),
  (TTimeField, TDBEdit),
  (TMemoField, TDBMemo),
  (TBlobField, TDBImage),      {just a guess}
  (TGraphicField, TDBImage));
function ConvertClass(FieldClass: TFieldClass) : TControlClass;
var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  for I := 1 to FieldTypeCount do
    if ConvertTable [I, 1] = FieldClass then begin
      ConvertClass := TControlClass (ConvertTable [I, 2]);
      Found := True;
      break; {jump out of for loop}
    end;
  if not Found then
    raise Exception.Create ('Match not found');
end;
```

➤ *Listing 4*

```
RowNum := 0;
for I := 0 to FieldList.Items.Count - 1 do
  if FieldList.Selected[I] then begin
    StringGrid1.Cells[0, RowNum] := Format ('%d) %s [%s]',
    [Table1.FieldDefs[I].FieldNo, Table1.FieldDefs[I].Name,
    ConvertClass(Table1.FieldDefs[I].FieldClass).ClassName]);
    StringGrid1.Cells[1, RowNum] := Table1.FieldDefs[I].Name;
    Inc(RowNum);
  end;
StringGrid1.RowCount := RowNum;
```

component is set using the name of the class of the control plus the name of the field. The problem here is that the names of the fields might include spaces or other special characters invalid for an identifier.

To solve this problem we've written a `NormalizeString` procedure, which replaces invalid characters with underscores. It also calls Delphi's `IsValidIdent` function, which checks whether a string is a valid Pascal identifier.

Now we need to set the `DataSource` and `DataField` properties of the data-aware components. The problem is that each VCL class representing a data-aware component defines these properties directly and does not inherit them from a common ancestor class. So we have no way to use inheritance, polymorphism or other language features to define these properties, but we have to write a chain of `if` statements to consider the various cases. Here is an example:

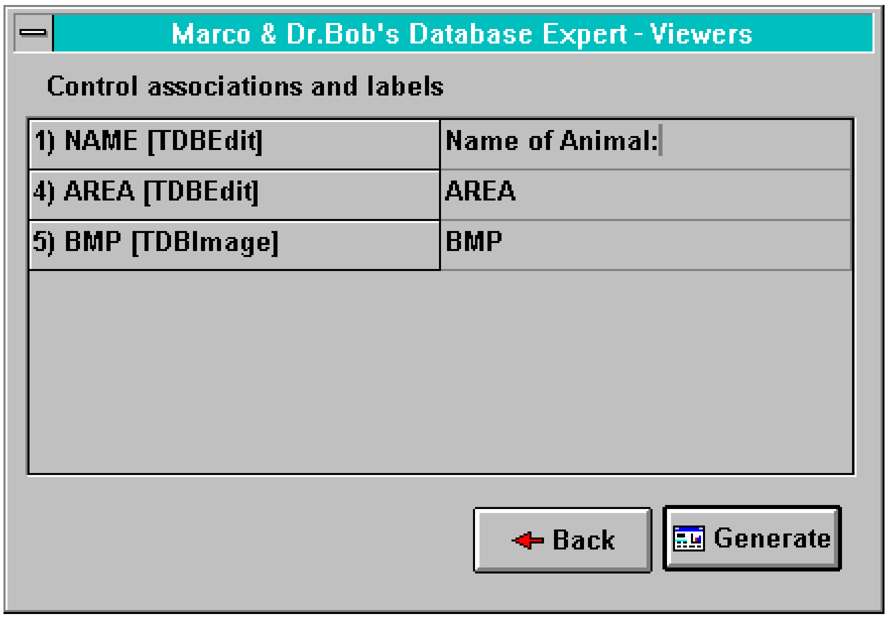
```
if CtrlClass = TDBEdit then
begin
  TDBEdit(NewDBComp).DataSource :=
   NewForm.DataSource1;
  TDBEdit(NewDBComp).DataField :=
    Table1.FieldDefs[I].Name;
```

You can actually write the cast as:

```
(NewDBComp as CtrlClass)
```

but this code returns a `TControl` and this class doesn't define the data related properties. The existence of the property is checked at compile time, so we can't use a dynamic data type instead. Unless we access the published properties of the object directly, we have to write some traditional code for this vital portion of the program.

At the end some more code is used to set the proper height of the form, making it as tall as possible but avoiding making it longer than the screen. The width, instead, is increased only where there is a scrollbar, adding the width of this last element (returned by the `GetSystemMetrics` API call). At the end the form is displayed to the user, as you can see in Figure 5.


Marco & Dr.Bob's Database Expert - Viewers

Control associations and labels

1) NAME [TDBEdit]	Name of Animal:
4) AREA [TDBEdit]	AREA
5) BMP [TDBImage]	BMP

➤ *Figure 4*

```
NewLabel := TLabel.Create(NewForm);
NewLabel.Parent := NewForm.ScrollBox1;
NewLabel.Name := 'Label' + IntToStr(I);
NewLabel.Caption := StringGrid1.Cells[1, RowNum];
NewLabel.Top := Y;
NewLabel.Left := 10;
NewLabel.Width := 130;
CtrlClass := ConvertClass(Table1.FieldDefs[I].FieldClass);
NewDBComp := CtrlClass.Create(NewForm);
NewDBComp.Parent := NewForm.ScrollBox1;
NewName := CtrlClass.ClassName + Table1.FieldDefs[I].Name;
NormalizeString(NewName);
NewDBComp.Name := NewName;
NewDBComp.Top := Y;
NewDBComp.Left := 150;
NewDbComp.Width := NewForm.ScrollBox1.Width - 160;
```

➤ *Listing 5*

Expert Advise?

The program so far (on the disk as DYNAVIEW.DPR) allows you to create database forms according to the parameters set in the notebook pages. This is very good as an example of a runtime application using databases, but it lacks the capability of a development tool. There is no way to save the structure of the forms you've created. This is the aim of the remainder of this article, which will also show you how to turn this program into a full-blown Delphi Expert.

Generating Source Code

First of all, we have to make some modifications in MAINFORM.PAS, in order to generate a synchronised Object Pascal source file for the form. Furthermore, we need to make sure that the form is not destroyed after it is shown. For this, we need to make modifications in the original code partially shown in Listing 5 for the response method of the `Generate` button. One of the changes is to eliminate the local variable `NewForm` and instead use the global variable `ResultForm` to hold the resulting form instance variable. This is the only way to ensure that the generated form instance would be re-usable outside the response method for the `Generate` button (after all, our Expert needs to get its hands on it to add it to the project).

Also, we need to generate the exact source code that would reflect the actual form that was created. This source code can be divided into three parts: the first part of the unit (unit header and initial definition of the `TResultForm` class), the last part of the unit (the end of the TResultForm class definition and the implementation section) and in between these two we have to enter the definitions for all the components (labels and data-aware controls) that are also created on-the-fly on the result form itself. Note that we use compiler directives in the final source code to make sure the code can be compiled for the DynaView application as well as the MarcoBob Database Expert. The first part of the unit is generated as shown in Listing 6 (f is a variable of type System.Text).

The last part of the unit is generated as shown in Listing 7 (note that we don't do a Show of the ResultForm any more; all we do is create it and make it ready to be used for the expert).

In between these two parts of the generated unit source code, we need to make sure that for each control which was created dynamically a corresponding variable declaration is entered in the source file. These modification are made in the original listing (Listing 5) at two places – see Listing 8.

Saving the Form

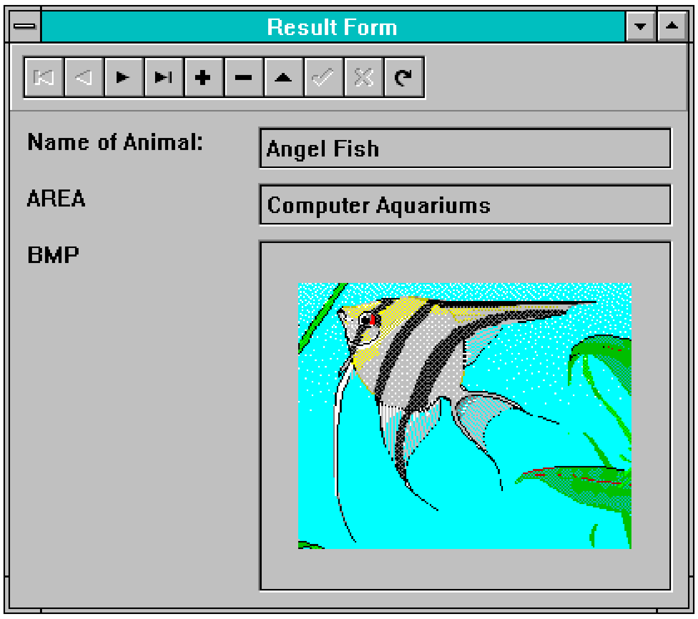
So, now we have the source code of the form itself generated on disk. But how do we generate a compatible .DFM file? We still have the resulting form (in the global ResultForm instance) and if we look hard enough in the Delphi Help and VCL source code we find that we can use the WriteComponentResFile method to write a class instance to file, which is exactly what we need.

Form Expert Interface

So far, we've focused on the execution of our Expert. There is more to an expert than the Execute method, as we've shown in Issue 3 of *The Delphi Magazine*. We need to derive our Database Expert (called TMarcoBobExpert) from the abstract base class TIExpert and override every method. The class definition of TMarcoBobExpert is shown in Listing 9.

For the implementation section we first need to override GetStyle and return esStandard (we could make it a form expert, but it's

➤ *Figure 5*



```
{$IFDEF EXPERT}
  {generate the first part of the unit source}
  System.Assign(f,UnitName+'.PAS');
  System.Rewrite(f);
  writeln(f,'unit ',ExtractFileName(UnitName),';');
  writeln(f,'interface');
  writeln(f,'uses');
  writeln(f,
  '  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,');
  writeln(f,'  Forms, Dialogs, DB, DBTables, DBCtrls, ExtCtrls;');
  writeln(f);
  writeln(f,'type');
  writeln(f,'  TResultForm = class(TForm)');
  writeln(f,'    Panel1: TPanel;');
  writeln(f,'    DBNavigator1: TDBNavigator;');
  writeln(f,'    ScrollBox1: TScrollBox;');
  writeln(f,'    DataSource1: TDataSource;');
  writeln(f,'    Table1: TTable;');
{$ENDIF}
```

➤ *Listing 6*

```
{$IFDEF EXPERT}
  writeln(f,'    procedure FormClose(Sender: TObject;
    var Action: TCloseAction);');
  writeln(f,'  private');
  writeln(f,'    { Private declarations }');
  writeln(f,'  public');
  writeln(f,'    { Public declarations }');
  writeln(f,'  end;');
  writeln(f);
  writeln(f,'var');
  writeln(f,'  ResultForm: TResultForm;');
  writeln(f);
  writeln(f,'implementation');
  writeln(f);
  writeln(f,'{$R *.DFM}');
  writeln(f);
  writeln(f,'procedure TResultForm.FormClose(Sender: TObject; '+
    'var Action: TCloseAction);');
  writeln(f,'begin');
  writeln(f,'  Action := caFree;');
  writeln(f,'end;');
  writeln(f);
  writeln(f,'end.');
  System.Close(f);
  ModalResult := mrOk
{$ELSE}
  ResultForm.Show;
{$ENDIF}
end;
```

➤ *Listing 7*

easier for now to leave it as a standard Help menu expert). The GetIDString and GetName always need to be overridden and return the name of our Database Expert. For a standard expert, GetGlyph and GetComment can return nothing, only GetMenuText and GetState are important. These define the text of the menu item (*Marco & Dr.Bob's Database Expert*) and the state (just enabled). The source code is shown in Listing 10.

Execute

We've left out one important detail of the expert so far: the Execute method. This is the method that, as the name indicates, gets executed whenever the expert is activated from the menu.

So, this is the place where we start the original application (the notebook with the four pages to indicate which alias, table, fields and label names we want). At the end of the Execute method, when the user has clicked on the Generate button, the source code for the unit will be generated and an instance of the corresponding form will be available in the global variable ResultForm. Before the expert gets to that point, however, it must first get a new unique filename for the new unit, a name that has to be given to the notebook code somehow, so that the dynamically generated unit source code can be put in the correct file. For this, we need to call a function from ToolServices which is called GetNewModuleName, which returns a unique filename for the unit as well as for the form itself (see the file MARCOBOB.PAS on the disk for more details on the Execute method).

When the Generate button on the notebook has been pressed and control returns to the expert we have a generated source file on disk and an instance of ResultForm on our hands. At that time, all the expert needs to do is to call a function called CreateModule from the ToolServices and add the newly generated form and unit to the project. Unfortunately, CreateModule only works with TIMemoryStreams and we have a file on disk and an instance in memory and no stream whatsoever.

This problem can be solved in three steps. First, we need to write the ResultForm instance to a disk file as well. We need to call WriteComponentResFile with the unique form filename as follows:

```
NewLabel := TLabel.Create(ResultForm);
NewLabel.Parent := ResultForm.ScrollBox1;
NewLabel.Name := 'Label' + IntToStr(I);
{$IFDEF EXPERT}
  writeln(f,'    Label',IntToStr(i),': TLabel;');
{$ENDIF}
NewLabel.Caption := StringGrid1.Cells[1, RowNum];
NewLabel.Top := Y;
NewLabel.Left := 10;
NewLabel.Width := 130;
CtrlClass := ConvertClass(Table1.FieldDefs[I].FieldClass);
NewDBComp := CtrlClass.Create(ResultForm);
NewDBComp.Parent := ResultForm.ScrollBox1;
NewName := CtrlClass.ClassName + Table1.FieldDefs[I].Name;
NormalizeString (NewName);
NewDBComp.Name := NewName;
{$IFDEF EXPERT}
  writeln(f,'    ',NewName,': ',CtrlClass.ClassName,';');
{$ENDIF}
```

➤ *Listing 8*

```
Type
  TMarcoBobExpert = class(TIExpert)
  public
    { Expert Style }
    function GetStyle: TExpertStyle; override;
    { Expert Strings }
    function GetIDString: string; override;
    function GetName: string; override;
    function GetGlyph: HBITMAP; override;
    function GetComment: string; override;
    function GetMenuText: string; override;
    function GetState: TExpertState; override;
    { Launch the Expert }
    procedure Execute; override;
  end;
```

➤ *Listing 9*

```
function TMarcoBobExpert.GetStyle: TExpertStyle;
begin
  Result := esStandard
end {GetStyle};
function TMarcoBobExpert.GetIDString: String;
begin
  Result := 'Marco.DrBob.Database.Expert'
end {GetIDString};
function TMarcoBobExpert.GetName: String;
begin
  Result := 'MarcoBob Expert'
end {GetName};
function TMarcoBobExpert.GetGlyph: HBITMAP;
begin
  Result := 0
end {GetGlyph};
function TMarcoBobExpert.GetComment: String;
begin
  Result := ''
end {GetComment};
function TMarcoBobExpert.GetMenuText: String;
begin
  Result := 'Marco && Dr.Bob''s Database Expert'
end {GetMenuText};
function TMarcoBobExpert.GetState: TExpertState;
begin
  Result := [esEnabled]
end {GetState};
```

➤ *Listing 10*

```
if (Form1.ShowModal = idOk)
then begin
  WriteComponentResFile(
    FormName, ResultForm);
  ResultForm.Close;
  ResultForm := nil;
```

After we've written the instance of ResultForm to disk there is no need to keep it around any longer, so we should close it (then it will free itself) and assign the global variable ResultForm to nil.

WriteComponentResFile can be used to write an instance of a component to a binary resource file. And guess what's inside a .DFM file? Indeed, it's the binary resource of a form. This also means that using ReadComponentResFile you can read just about any .DFM file and use it to generate the corresponding form on-the-fly (that should give you some new ideas to experiment with...).

So, now we have two disk files: one for the unit source code in a .PAS file and one for the corresponding form in a .DFM file.

At this point, we could start up Delphi's Project Manager and simply add the unit to the project. However, we want to automate things, so we need to do some more work.

How do we get a file on disk into a TIMemoryStream? We could think of only one way: read the file into a TFileStream, then copy the TFileStream into a TMemoryStream and finally use the TMemoryStream to create a TIMemoryStream (which is then said to own the stream itself). This may sound very complex and inefficient, but it turns out that it doesn't really take that much time. For now, it works and allows us to get a disk file into a TIMemoryStream to be used later as an argument to CreateModule. See Listing 11.

Now, finally, we're able to call ToolServices.CreateModule which will create a new module based on the name of the unit, the TIMemoryStream image of the unit source code, the form binary and finally some options which specify that this unit/form is to be added to the project and that we want to show the unit as well as the form:

```
ToolServices.CreateModule(
  UnitName, IUnitStream,
  IFormStream, [cmAddToProject,
  cmShowSource, cmShowForm])
```

The complete source code for the Execute function is on the disk.

Installing this new Database Expert is just like installing a new component. Just pick Options | Install Component from the menu, and enter the file MARCOBOB.PAS (note: you need to have MAINFORM.PAS and MAINFORM.DFM in the same directory in order to be able to compile the expert). After rebuilding COMPLIB.DCL, our expert will be part of Delphi itself and can be found on the Help menu.

We can now generate the same form as before and if we compile the program, then we get the same form as can be seen in Figure 5. However, there is one important difference: this time, we have also generated the source code and .DFM file for the form. We can now make any changes we want, or generate a dozen more forms and add them all to our project. *Marco & Dr.Bob's Database Expert* really works! The full source code is included on this month's disk.

32-bit Version

By the time you read this article, Delphi 2.0 will probably be just available. So, you might well wonder if our expert is compatible with Delphi 2.0. Will it run on Windows 95? Well, so far it seems that the promise Borland made about source compatibility is true: we managed to get our Database Expert running in the pre-release version of Delphi 2.0 without a single code modification! And if you want to change anything in this expert, then remember that this time you've got full source code!

Acknowledgements

Marco and Dr.Bob's Database Expert was the result of a wild "live" hacking session during the November 1995 Delphi seminar of the UK Delphi Developers Group (for details of the group send email to CompuServe 100016,355) which started in Manchester – with 'almost' success (!) – and ended in London with a working initial version. This article is the spin-off of that great adventure, for which we want to thank Joanna Pooley and Phil Goulson from the bottom of our hearts! We'd love to do it again someday...

Marco Cantu is the author of *Mastering Delphi*, published by Sybex, and is currently working on a new edition of the book covering Delphi 2.0. He lives in Italy, where he teaches Delphi programming courses, and can be reached at CompuServe 100273,2610

Bob Swart (email: CompuServe 100434,2072) is co-author of *The Revolutionary Guide to Delphi 2.0*, soon to be published by WROX Press, columnist for The Delphi Magazine and professional software developer for Bolesian BV in The Netherlands. In his spare time, Bob likes to watch video tapes of Star Trek Voyager with his 2-year old son Erik Mark Pascal.

```
{ copy form data to a IMemoryStream }
MemoryStream := TMemoryStream.Create;
FileStream := TFileStream.Create(FormName, fmShareDenyNone);
repeat until MemoryStream.Write(Buffer,FileStream.Read(Buffer,Size)) = 0;
FileStream.Free;
MemoryStream.Position := 0;
IFormStream := TIMemoryStream.Create(MemoryStream);
IFormStream.OwnStream := True;
{ copy unit data to a IMemoryStream }
MemoryStream := TMemoryStream.Create;
FileStream := TFileStream.Create(UnitName, fmShareDenyNone);
repeat until MemoryStream.Write(Buffer,FileStream.Read(Buffer,Size)) = 0;
FileStream.Free;
MemoryStream.Position := 0;
IUnitStream := TIMemoryStream.Create(MemoryStream);
IUnitStream.OwnStream := True;
```

➤ *Listing 11*