Writing Your Own Experts

by Bob Swart

Delphi is an open development environment, in that it has interfaces to enable you to integrate your own tools and experts with it. This article will focus on writing and integrating new Experts with Delphi.

There are three kinds of experts: project, form and standard. The first two can be found in the `Options | Gallery` dialog, while standard experts are on the Help menu (like the Database Form Expert).

Project and form experts can be activated whenever you create a new project or form (just like project and form templates). Standard experts generally do not create a new project or form, but just a new file, or unit. A project expert develops an entire project for you based on your specific preferences. A form expert develops custom forms that are added to your current project.

These example experts are not external tools that can be started from Delphi, they actually communicate with Delphi and form an integral part of the development environment. While this is not so strange for the existing Delphi experts (after all, they were developed and added by the same team that developed Delphi in the first place), it sounds intriguing at least to know that we, too, can write a Delphi expert that is able to communicate with Delphi in the same way. Could we write an expert that also opens files in the IDE, and can start a new project from scratch? Yes, all this is possible, and more, as we will see shortly!

TIExpert

The major reason why everybody thinks experts are difficult is because they are not documented. Not in the manuals or on-line Help, that is. If you take a look at the documentation and source code on your hard disk, though, you'll find some important files and even two example experts. The key example files can be found in the DELPHI\DOC subdirectory and are EXPTINTF.PAS and TOOLINTF.PAS. The first one shows how to derive and register our own Expert, while the second one shows how to use the tool serviees of Delphi to make the integration complete.

If we want to derive our own expert, say `TMyFirstExpert`, we have to derive it from the abstract base class `TIExpert`, which has seven abstract member functions (`GetStyle`, `GetName`, `GetComment`, `GetGlyph`, `GetState`, `GetIDString` and `GetMenuText`) and one member procedure (`Execute`).

My First Expert: TMy1stExp

Let's have a closer look at our first expert from Listing 1. Since `TIExpert` is an abstract base class, we need to override every function. First of all, we need to specify the style of the expert with the `GetStyle` method that can return one of three possible values: `esStandard` to tell the IDE to treat the interface to this expert as a menu item on the `Help` menu, `esForm` to tell the IDE to treat this expert interface in a fashion similar to form templates, or `esProject` to tell the IDE to treat this interface in a fashion similar to project

➤ *Listing 1 Source Code for My First Expert (MY1STEXP.PAS)*

```
unit My1stexp;
interface
uses
  WinTypes, Dialogs, ExptIntf;
Type
  TMy1stExp = class(TIExpert)
  public
    function GetStyle: TExpertStyle; override;  { Style }
    { Expert Strings }
    function GetName: string; override;
    function GetComment: string; override;
    function GetGlyph: HBITMAP; override;
    function GetState: TExpertState; override;
    function GetIDString: string; override;
    function GetMenuText: string; override;
    procedure Execute; override;  { Launch the Expert }
  end;
  procedure Register;
implementation
function TMy1stExp.GetStyle: TExpertStyle;
begin
  Result := esStandard
end;
function TMy1stExp.GetName: String;
begin
  Result := 'My First Expert'
end;
function TMy1stExp.GetComment: String;
begin
  Result := '' { not needed for esStandard }
end;
function TMy1stExp.GetGlyph: HBITMAP;
begin
  Result := 0 { not needed for esStandard }
end;
function TMy1stExp.GetState: TExpertState;
begin
  Result := [esEnabled]
end;
function TMy1stExp.GetIDString: String;
begin
  Result := 'DrBob.MyFirstExpert'
end;
function TMy1stExp.GetMenuText: String;
begin
  Result := '&My First Delphi Expert...'
end;
procedure TMy1stExp.Execute;
begin
  MessageDlg('Hello World: My First Expert is alive!',
             mtInformation, [mbOk], 0)
end;
procedure Register;
begin
  RegisterLibraryExpert(TMy1stExp.Create)
end;
end.
```

templates. For our TMy1stExp, a *standard* type expert that shows a MessageDlg to indicate it is alive, we can use the esStandard style.

After we've set the style of the expert, all we need to do is fill in the other options accordingly. GetName must return a unique descriptive name identifying this expert, like 'My First Expert'. If style is esForm or esProject then GetComment should return a short sentence describing the function of this expert. Since the style is esStandard, we can return an empty string. If style is esForm or esProject then GetGlyph should return a handle to a bitmap to be displayed in the form or project list boxes or dialogs. This bitmap should have a size of 60x40 pixels in 16 colours. Again, since the style is esStandard, we can return 0 here. If the style is esStandard then GetState returning esChecked will cause the menu to display a checkmark. This function is called each time the expert is shown in a menu or listbox in order to determine how it should be displayed. We just leave it esEnabled for now. The GetIDString should be unique to each expert. By convention, the format of the string is: CompanyName.ExpertFunction. If the style is esStandard then GetMenuText should return the actual text to display for the menu item, like 'My First Delphi Expert'. Since this function is called each time the parent menu is pulled down, it is even possible to provide context sensitive text.

Finally, the Execute method is called whenever this expert is invoked via the menu, form gallery dialog, or project gallery dialog. The style will determine how the expert was invoked. In this case, we just call a MessageDlg in the Execute method to indicate that the expert is actually alive.

To install our first expert, all we need to do is act like it's a new component: pick Options | Install and add it to the list of installed components. When Delphi is done with compiling and linking COMPLIB.DCL, you can find our first new expert in the Help menu. Just click on it and it will show that it's alive (see Figure 1).

➤ *Figure 1*
My First Delphi Expert is alive!

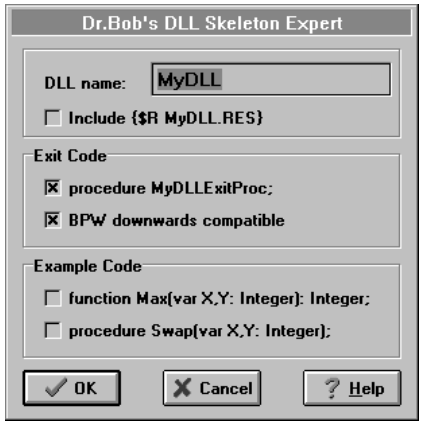
And Now For Something Completely Different...

Now that we've seen our first nice, but useless, Delphi expert, it's time to move on to more serious matters. I want to make a little sidestep to a subject that will make a good example of a more serious Delphi expert.

On the CompuServe DELPHI forum, one of the queries that comes up rather frequently is "*How do I write a DLL with Delphi?*". The answer is not just that you need to write the code starting with library and so on, the answer also needs to explain how to compile the source for a DLL with Delphi. In their wisdom, Borland made the Delphi IDE only capable of compiling the current project. If you just open a single file with the source for the DLL and press Ctrl-F9 to compile it, you won't get what you want. You must actually open your DLL source file as a *project* and then you can compile your DLL. During this process, Delphi will generate .OPT and .RES files if these don't already exist. All things considered, I would like something that enables me to open a new or existing DLL source file at once so I can compile it.

Speaking of DLLs, whenever I sit down to write a DLL in Delphi (or Borland Pascal, for that matter), I pick up an old one to use as skeleton. Mostly, I re-use the setup for the ExitProc routine and the exports settings. For this purpose, I've written a DLL skeleton that can be loaded every time I need it. Considering the fact that some of my friends also use this skeleton for their new DLLs, I decided to make it something truly re-usable: a Delphi DLL Skeleton Generator (see Figure 2).

As you can see, I've included the key functionality all in one Form: *How do I write a Resource-only DLL? How do I write my own WEP* (the



➤ *Figure 2*
Delphi DLL Skeleton Generator

same as ExitProc)? *How do I export routines from a DLL?* All these questions can be answered if you just select the appropriate options and click OK to generate the DLL skeleton source code. A sample skeleton DLL with all options enabled, except BPW compatibility (which does not include the SysUtils unit and AddExitProc routine but requires you to setup the ExitProc chain by hand), can be found in Listing 2.

Behind the OkButtonClick is the source code generator that writes the selected source code to file. Now I want something like this integrated into Delphi itself, so I can generate a new Delphi DLL Skeleton and open it as my new project at the same time. In order to make the DLL Skeleton Generator a Delphi expert, all we have to do is connect our expert Execute method with our DLL Skeleton Generator Form, as in Listing 3.

So, whenever the expert is executed, it will see if our DLL Skeleton Generator Form already exists (ie if the expert is already being executed) and create it if it doesn't exists. It will then show the form and give it the input focus. The DLL Skeleton Generator Form is then in control.

The Final Frontier...

Only one thing remains: the final integration with the Delphi IDE. I would like to be able open a new project with the source of the generated DLL Skeleton inside. For this, we need to communicate with the Delphi IDE itself. This is

possible with the special ToolServices that are provided from Delphi to its experts. Like the expert interface, the ToolServices are not documented in the manual or on-line help. The only place you can find more information on this is in the TOOLINTF.PAS file, again in the DELPHI\DOC directory.

First of all, we need to check if the ToolServices are available to us. This is just a check to see if ToolServices (a global variable from the TOOLINTF unit) is not nil. If ToolServices are available, we can do several things. I would like to close the current project, which can be done with the function ToolServices.CloseProject. Then, I would like to open a new project, with the generated DLL Skeleton source file as the filename, which can be done with the function ToolServices.OpenProject.

The last part of the OkButtonClick method of the DLL Skeleton Generator Form is therefore as shown in Listing 4.

Simple, eh? That's all we need to communicate with Delphi and write a truly integrated Delphi standard expert.

Project Expert

The DLL Skeleton Generator Expert is still a standard expert, only accessible from the Help menu. I would like to make it a project expert, so we can select it when we start a new project. To do this, we have to derive the project expert from the standard expert and override four methods. First of all, we have to override GetStyle and return esProject. Also, we need to return a comment (this is not really needed) and a bitmap to display the expert in the Gallery.

Standard And Project?

Remember the Database Form Expert? This can be found in the Gallery as a form expert *and* in the Help menu as a standard expert. It seems to be both.

I would like to be able to use my DLL Skeleton Generator Expert not only as a standard expert but also as a project expert. In that case I have to modify the expert functions from Listing 1 to include both the esStandard and esProject styles (the result is in Listing 5). Also, GetIDString needs to return unique ID strings for *both* the standard and the project expert. Even though the two are essentially the same, I need to return two special IDs. If you don't, Delphi will just GPF when you try to install the experts. Which leads back to Rule #1 from the *Under Construction* column: always have a backup of COMPLIB.DCL at hand when you start to play with components and experts.

Now, if we install the expert, as before, we get both a standard expert in the Help menu and the project expert in the Gallery (see Figure 3). If we enable the gallery from the environment options, we can generate and open a DLL Skeleton source file every time we start a new project.

```
library MyDLL;
{ Generated by DLL Skeleton Expert (c) 1995 by Dr.Bob for The Delphi Magazine }
uses WinTypes, WinProcs, SysUtils;
{$R MyDLL.RES}
function Max(X,Y: Integer): Integer; export;
begin
  if X > Y then Max := X
    else Max := Y
end {Max};
procedure Swap(var X,Y: Integer); export;
var Z: Integer;
begin
  Z := X;
  X := Y;
  Y := Z
end {Swap};
exports max index 1,
        swap index 2;
procedure MyDLLExitProc; far;
begin
  { WEP & cleanup }
end;
begin
  AddExitProc(MyDLLExitProc);
end.
```

➤ *Listing 2 Generated DLL skeleton source code*

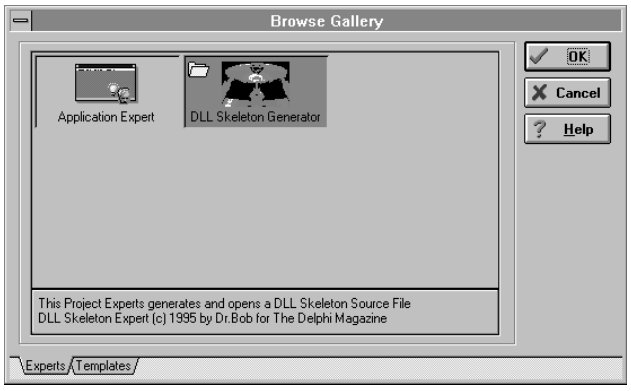
```
procedure TDLLSkExp.Execute;
begin
  if not Assigned(DLLSkeletonGenerator) then
    DLLSkeletonGenerator := TDLLSkeletonGenerator.Create(Application);
  DLLSkeletonGenerator.Show;
  DLLSkeletonGenerator.SetFocus
end;
```

➤ *Listing 3*

```
if ToolServices <> nil then begin
  { I'm an expert!! }
  if ToolServices.CloseProject then
    ToolServices.OpenProject(ExtractFileName(DLLName.Text)+'.PAS')
end
```

➤ *Listing 4*

➤ *Figure 3 DLL Skeleton Generator installed ready for use*



```
unit Dllskexp;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Buttons, StdCtrls,
  ExptIntf, ToolIntf;
{ definition of class TDLLSKeletonGenerator is on disk}
Type
  TDLLSkeletonStandardExpert = class(TIExpert)
  public
    { Expert Style }
    function GetStyle: TExpertStyle; override;
    { Expert Strings }
    function GetIDString: string; override;
    function GetName: string; override;
    function GetComment: string; override;
    function GetGlyph: HBITMAP; override;
    function GetState: TExpertState; override;
    function GetMenuText: string; override;
    procedure Execute; override;    { Launch the Expert }
  end;
  TDLLSkeletonProjectExpert =
    class(TDLLSkeletonStandardExpert)
  public
    { Expert Style }
    function GetStyle: TExpertStyle; override;
    { Expert Strings }
    function GetIDString: string; override;
    function GetComment: string; override;
    function GetGlyph: HBITMAP; override;
  end;
  procedure Register;
implementation
{$R *.DFM}
{ class TDLLSKeletonGeneratorm implementation is on disk}
function TDLLSkeletonStandardExpert.GetStyle:
  TExpertStyle;
begin
  Result := esStandard
end;
function TDLLSkeletonStandardExpert.GetIDString: String;
begin
  Result := 'DrBob.StandardDLLSkExp'
end;
function TDLLSkeletonStandardExpert.GetComment: String;
begin
  Result := '' { not needed for esStandard }
end;
function TDLLSkeletonStandardExpert.GetGlyph: HBITMAP;
begin
  Result := 0 { not needed for esStandard }
end;
function TDLLSkeletonStandardExpert.GetName: String;
begin
  Result := 'DLL Skeleton Generator'
end;
function TDLLSkeletonStandardExpert.GetState:
  TExpertState;
begin
  Result := [esEnabled]
end;
function TDLLSkeletonStandardExpert.GetMenuText: String;
begin
  Result := 'Dr.&Bob''s DLL Skeleton Expert...'
end;
procedure TDLLSkeletonStandardExpert.Execute;
begin
  if not Assigned(DLLSkeletonGenerator) then
    DLLSkeletonGenerator :=
      TDLLSkeletonGenerator.Create(Application);
  DLLSkeletonGenerator.Show;
  DLLSkeletonGenerator.SetFocus
end;
{$R DLLSKEXP.RES}
Const DLLSKEXPBITMAP = 666; { Bitmap ID }
function TDLLSkeletonProjectExpert.GetStyle:
  TExpertStyle;
begin
  Result := esProject
end;
function TDLLSkeletonProjectExpert.GetIDString: String;
begin
  Result := 'DrBob.ProjectDLLSkExp'
end;
function TDLLSkeletonProjectExpert.GetComment: String;
begin
  Result := 'This Project Experts generates and opens '+
    'a DLL Skeleton Source File'#13+ 'DLL Skeleton '+
    'Expert (c) 1995 by Dr.Bob for The Delphi Magazine';
end;
function TDLLSkeletonProjectExpert.GetGlyph: HBITMAP;
begin
  Result := LoadBitMap(HInstance,
              MakeIntResource(DLLSKEXPBITMAP))
end;
procedure Register;
begin
  RegisterLibraryExpert(
    TDLLSkeletonStandardExpert.Create);
  RegisterLibraryExpert(
    TDLLSkeletonProjectExpert.Create);
end;
end.
```

➤ *Listing 5 DLL Skeleton Generator Standard and Project Expert*

If we select the DLL Skeleton Expert, we can then select the required options (as in Figure 2). If we click on OK, the expert closes and we're in our main project: the generated source of the DLL.

Since the generated DLL source code is opened as a new project, we can instantly compile it by pressing Ctrl-F9. And once you have a skeleton DLL, it's easy to build on it and add your own functions.

Serious Business...

The example DLL Skeleton Generator Expert is included on the subscribers' disk with this issue. You'll also find another expert, the one I wrote about in the last issue: *HeadConv*. This solves a more serious problem that many people have: "*How do I use this foreign DLL written in C, as I only have the C header file with it and no Delphi import unit?*" The answer is to convert the C DLL header file to a Delphi import unit. This is no simple task, especially for large header files, and my *HeadConv C DLL Header Converter Expert* tries to assist in this task by creating an initial conversion from which to start. For some headers, the initial conversion is good enough, for others extra work might be needed. Specifically, the declaration of nested structs and actual code (as opposed to function declarations) will be a source of problems (pun intended).

I've decided to sell *HeadConv* as a shareware tool. The version on the disk is fully functional, but for a registration fee of $25 you get a more advanced version with explicit import unit capabilities and the source code of the expert (but not of the parser). The CompuServe SWREG forum registration ID is 6533). See the advert in this issue for more details.

Bob Swart is a professional software developer using Borland Pascal, C++ and Delphi; email: 100434.2072@compuserve.com