Extending the Lazarus IDE: Custom forms and units

Michaël Van Canneyt

July 29, 2005

Abstract

This article explores new ways to extend the 'New' dialog in the Lazarus IDE. It shows how to add custom form classes to the IDE, and how the 'New' dialog can be divided in categories. But it starts by offering a new (and simpler) way to create a project from a template, which was discussed in a previous article.

1 Introduction

In a previous article, it was shown how to use the various interfaces in the Lazarus IDE to implement project templates. The code was written so it would show as much of the IDE interfaces as possible. In this article, it will be shown how the same functionality can be achieved with less code by using a simple call in the IDE. Indeed, the call was actually trivially extended by the Lazarus maintainers based on a suggestion by the author while writing the previous article, thus showing once more one of the major advantages of Open Source projects: instant reaction on suggestions and bug reports.

As a second item, an alternate way to present the templates will be discussed: The New dialog can be divided in categories. A new category 'Template Projects' will be created, and the various templates will be presented as 'New' items in that category.

Last but not least, it will be shown how it is possible to add custom properties to a TForm descendant, which can be used as a base for new forms, allowing the developer to manipulate these new properties.

This article is a continuation of a previous article, it may be necessary to reread the previous article to re-familiarize with the concepts introduced there. The code of the previous article is present on the disk (and the text of the article as well, in PDF format).
Joerg, is this possible ? if not, remove the last part of this sentence.

2 The template projects wizard revisited

In the previous article, it was shown how to implement template projects: Create a new project, based on a copy of an existing project, with automatic variable expansion. To do this, 2 classes were needed:

```
\item[TTemplateProjectDescriptor] a descendent from
\var{TProjectDescriptor}, which tells the IDE how to start a new template
```

```
project.
\item[TProjectFileDesc] a descendent of \var{TProjectFileDescriptor}, which
tells the IDE how to create the various files, present in the template. For
each file in the template, an instance of \var{TProjectFileDesc} was
created.
```

After some discussion with one of the Lazarus maintainers, it became clear that a trivial enhancement to an existing lazarus call, made the whole `TProjectFileDesc` class superfluous.

The `DoOpenEditorFile` call of the `TLazIDEInterface` can be used to open a file on disk, and add it to the project. This call is defined as follows:

```
function DoOpenEditorFile(AFileName:string;
                          PageIndex: integer;
                          Flags: TOpenFlags): TModalResult;
```

The meaning of the `AFileName` parameter should be obvious. The `PageIndex` tells the IDE on what editor page it should open the file, but is commonly set to -1, to indicate that a new page must be created. The `Flags` is one or more of the following flags:

ofProjectLoading the file is opened as part of opening a whole project.

ofOnlyIfExists Do not create an empty file if the file does not exist.

ofRevert Reload the file if it is already open in the IDE.

ofQuiet Display less messages

ofAddToRecent add the filename to recent files

ofRegularFile open as a regular file (a unit).

ofConvertMacros replace macros in filename

ofUseCache do not update file from disk if the file is in the file cache.

ofMultiOpen Can be set during loading of multiple files, for speed readons.

ofDoNotLoadResource do not open an associated form file.

ofAddToProject Add the file to the current project (if it exists)

The last flag is what is needed to simplify the template project descriptor.

As explained in the previous article, when a new project is started, the `CreateStartFiles` function is called to create any initial files belonging to the project. This function was previously implented using a `TProjectFileDesc` class, but can now be implemented much simpler using the `DoOpenEditorFile` call:

```
Function CreateStartFiles(AProject: TLazProject): TModalresult;

Var
  I : Integer;
  E,FN : String;

begin
  if Assigned(FTemplate) then
```

```
    begin
    Result:=mrOK;
    For I:=0 to FTemplate.FileCount-1 do
      begin
      FN:=FTemplate.FileNames[I];
      E:=ExtractFileExt(FN);
      If (CompareText(E,'.lpr')<>0)
         and (CompareText(E,'.lfm')<>0) then
        begin
        FN:=FProjectDirectory+FTemplate.TargetFileName(FN,FVariables);
        LazarusIDE.DoOpenEditorFile(FN,-1,[ofAddToProject]);
        end;
      end;
    end
  else
    Result:=mrCancel;
end;
```

The `FTemplate` is searched for all files it contains, and all pascal source files are simply added to the project using the `DoOpenEditorFile` call. Any associated .lrs or `.lfm` files will be detected by the IDE and loaded as well.

When using the `TProjectFileDesc` class, the new project files were created in code by reading the template files, expanding any variables found, and passing the resulting source text directly to the IDE.

When using the `DoOpenEditorFile` call, the files must be present on disk. This means that all the files in the template must have been copied already before the call to `CreateStartFiles`. The correct place to copy the template files is when the project descriptor is initialized, after the user has provided all needed information, in the `DoInitDescriptor` call:

```
function TTemplateProjectDescriptor.DoInitDescriptor: TModalResult;

begin
  InitTemplates;
  Result:=ShowOptionsDialog;
  If (Result=mrOK) and (FVariables.Count<>0) then
    Result:=ShowVariableDialog;
  If (Result=mrOK) then
    begin
    FVariables.Values['ProjName']:=FProjectName;
    FVariables.Values['ProjDir']:=FProjectDirectory;
    FTemplate.CreateProject(FProjectDirectory,FVariables);
    end;
end;
```

As explained in the previous article, the `CreateProject` call from the `TTemplate` class will copy all files to the indicated directory.

After these little changes, there is no more need for the `TProjectFileDesc` class, and it can be removed from the unit altogether.

3 Creating item categories

The lazarus **'File|New'** dialog contains by default 3 categories:

File Any file descriptor registered with the `RegisterFileDescriptor` will be placed under this category.

Project Any project file descriptor registered with the `RegisterProjectDescriptor` call will be placed under this category.

Package Any package file descriptor RegisterPackageDescriptor will be placed under this category.

To understand how the IDE places a file descriptor in the IDE menu, we can examine the `RegisterFileDescriptor` call:

```
procedure RegisterProjectFileDescriptor
   (FileDesc: TProjectFileDescriptor);
var
  NewItemFile: TNewItemProjectFile;

begin
  ProjectFileDescriptors.RegisterFileDescriptor(FileDesc);
  if FileDesc.VisibleInNewDialog then
    begin
    NewItemFile:=TNewItemProjectFile.Create(FileDesc.Name,
                                            niifCopy,
                                            [niifCopy]);
    NewItemFile.Descriptor:=FileDesc;
    RegisterNewDialogItem(FileDescGroupName,NewItemFile);
    end;
end;
```

All items in the 'New' dialog are described using a `TNewIDEItemTemplate` class, described in the newitemintf unit. The `TNewItemProjectFile` class used above is a descendent of this class.

So the IDE actually uses 4 kinds of classes to manage the File|New menu:

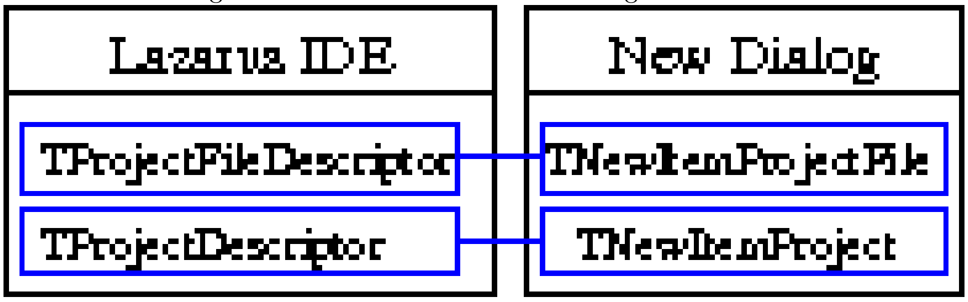
1. File description classes, which describe the contents of a file to be generated.
2. Project descriptions classes, which describe how to generate a new project.
3. A class which creates the visual representation of the description classes in the New dialog (`TNewIDEItemTemplate`).
4. A category class which helps in creating the visual representation. (`TNewIDEItemCategories`).

The former 2 classes are part of the projectintf unit, the latter two are part of the newitemintf unit. The details of these classes are not really important, they serve mainly as data storage. The classes are visualized in figure 1 on page 5

Adding a new category is done with the `RegisterNewItemCategory` call:

```
rocedure RegisterNewItemCategory(const ACategory: String);
begin
  NewIdeItems.Add(ACategory);
end;
```

Figure 1: The 4 IDE classes for adding custom files



This will create a new `TNewIDEItemCategories` instance, and add it to the list of categories.

The list of **'File|New'** commands is maintained in the `NewIDEItems` instance in the `newitemintf` unit. The `RegisterNewDialogItem` call adds a new item to this list:

```
procedure RegisterNewDialogItem(const Paths: string;
                                NewItem: TNewIDEItemTemplate);
begin
  NewIDEItems.RegisterItem(Paths,NewItem);
end;
```

The `Paths` string describes the category in which the new item is added. This is simply the name of the category (later, a hierarchical notion may be introduced). The `RegisterItem` simply looks for the category named `Paths` and attaches the `NewItem` instance to it.

So, when enhancing the **'File|New'** dialog, 2 things must be done:

1. Create a category with the `RegisterNewItemCategory` call.
2. Register project or file descriptor items in this category with the `RegisterProjectFileDescriptor` or `RegisterProjectDescriptor` calls.

4 Reorganizing the templates

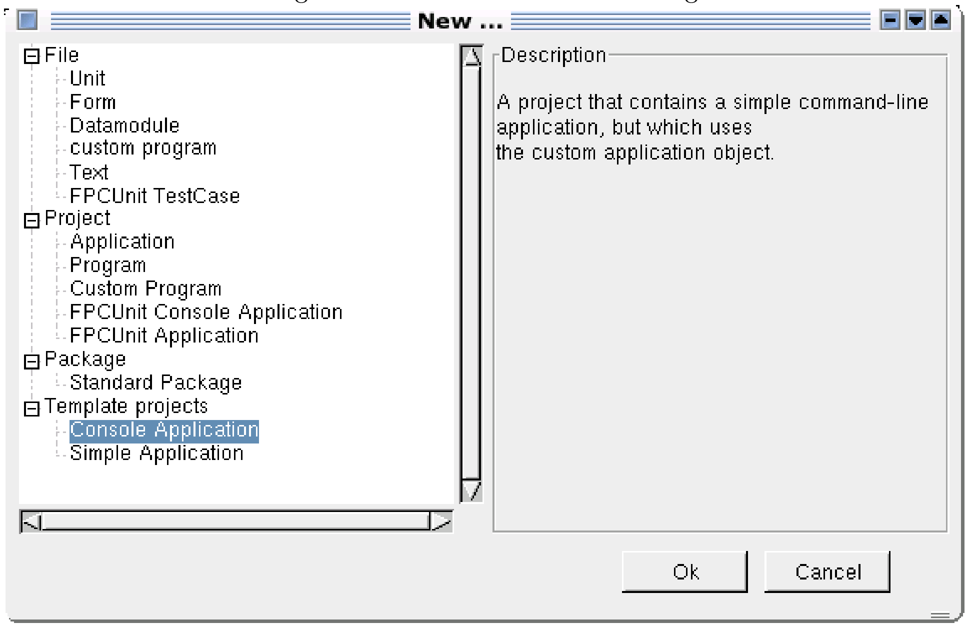
In the first version of the `Template projects` package, there was a single item under the 'Projects' category: 'Template projects'. After this item was chosen, a dialog was presented in which a template could be chosen. After the template was chosen, a dialog was presented which allowed to supply values for additional variables.

Using categories, it is possible to reorganize this: A 'Template Projects' category is made in the 'New' dialog, and the available templates will be presented as items below this category. The result should look something like in figure 2 on page 6.

To do this, the templates must be reorganized a bit. First of all, when the IDE starts, the available templates must be scanned, so they can be registered directly in the IDE. The best place to do this is in the `Register` procedure of the package:

```
Const
  STemplateCategory = 'Template projects';
```

Figure 2: The reworked 'New' dialog



A project that contains a simple command-line application, but which uses the custom application object.

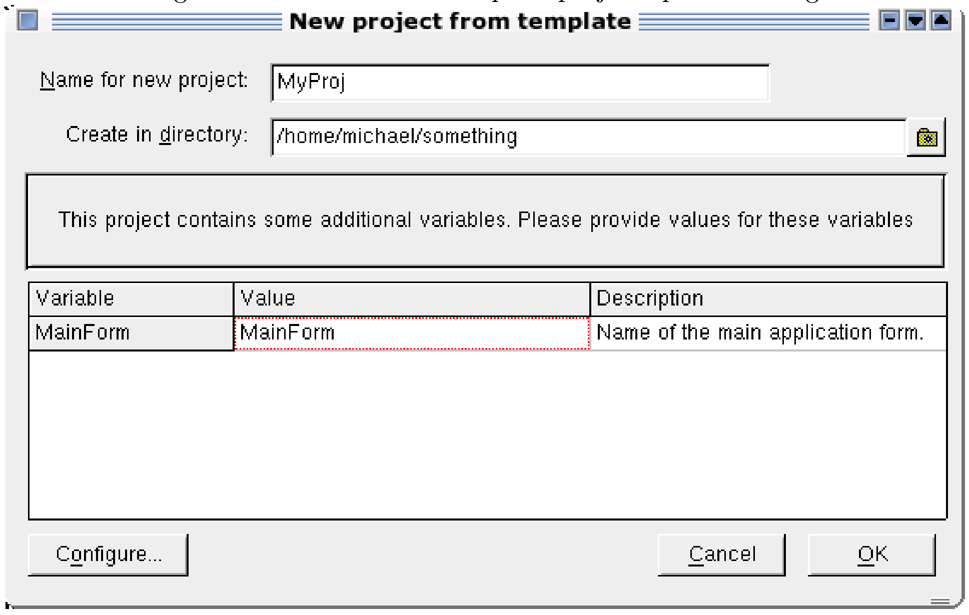
```
procedure Register;

Var
  I : Integer;
  D : String;

begin
  D:=GetTemplateDir;
  IDETemplates:=TProjectTemplates.Create(D);
  RegisterNewItemCategory(STemplateCategory);
  For I:=0 to IDETemplates.Count-1 do
    RegisterTemplateProject(IDETemplates[i]);
end;
```

After the templates are registered, a new category is registered. Then, for each available IDE template, a project descriptor is registered using the `RegisterTemplateProject` call:

```
procedure RegisterTemplateProject(ATemplate : TProjectTemplate);

var
  ProjDesc: TTemplateProjectDescriptor;

begin
  ProjDesc:=TTemplateProjectDescriptor.Create(Atemplate);
  RegisterProjectDescriptor(ProjDesc,STemplateCategory);
end;
```

For each project template, a separate instance of the project descriptor `TTemplateProjectDescriptor` is created. It is then registered with the standard `RegisterProjectDescriptor` call under the correct category.

Figure 3: The reworked 'Template project options' dialog



Name for new project: MyProj

Create in directory: /home/michael/something

This project contains some additional variables. Please provide values for these variables

Variable	Value	Description
MainForm	MainForm	Name of the main application form.

The `TTemplateProjectDescriptor` class, introduced previously, needs some changes: it no longer needs to offer the user a list of templates, instead, the template is passed to it when it is created:

```
onstructor TTemplateProjectDescriptor.Create(ATemplate : TProjectTemplate);
begin
  inherited Create;
  FTemplate:=ATemplate;
  Name:=FTemplate.Name
  FVariables:=TStringList.Create;
end;
```

An instance to the tempate is saved, and the name of the descriptor is set to the name of the template, so it will be displayed correctly in the 'New' dialog.

When the descriptor is initialized, only one dialog needs to be shown, where the name of the new project is chosen, the directory where to create the project, and the variables that need to be subsituted. The dialog looks like figure 3 on page 7. The new `DoInitDescriptor` function looks like this:

```
function TTemplateProjectDescriptor.DoInitDescriptor: TModalResult;

begin
  Result:=ShowOptionsDialog;
  If (Result=mrOK) then
    FTemplate.CreateProject(FProjectDirectory,FVariables);
end;
```

Which is much more simple than the version presented earlier in the article. The `ShowOptionsDialog` function is omitted. The interested reader can consult the source code that comes with this article.

5 Custom Forms

The file descriptors can also be used to register custom classes with the IDE. In principle, any TPersistent descendent (a class with RTTI) can be rehistered in the 'New' dialog. Of course, the IDE only knows how to display a `TDataModule` descendent or a `TCustomForm` descendent.

There are several reasons for wanting to create a custom form. One reason is to add additional behaviour to the all forms that should be created in an application , the other is to add or remove some of the published properties of `TForm`.

Now, to register a custom form, a `TProjectFileDescriptor` descendent class must be made. Lazarus already implements 2 descendents of this class, which are almost suitable for this purpose.

The first of these classes is `TFileDescPascalUnit`. It overrides the `CreateSource` method to create an empty unit. The form of the unit is controlled by the following cirtual methods:

```
function GetInterfaceUsesSection: string;
function GetInterfaceSource(const Filename, SourceName;
                            ResourceName: string): string;
function GetImplementationSource(const Filename, SourceName,
                                 ResourceName: string): string;
```

The meaning of these functions should be clear:

InterfaceUsesSection This function should return a string with the units that should go in the `Interface Uses` clause (it returns the `Classes` and `SysUtils` units by default).

GetInterfaceSource This function should return any other code that goes in the interface section. The parameters to this function should be obvious.

GetImplementationSource This method does the same as the previous one, but for the implementation section of the unit.

By default, both these methods return the empty string, i.e. no actual source is present. This class is used to create a new, empty, unit.

The second class can be used when generating units that need a form file: `TFileDescPascalUnitWithResource`. This descendent from `TFileDescPascalUnit` overrides some of the methods:

GetInterfaceSource This function will return an empty class declaration, using the resourcename for the class name.

GetImplementationSource Here the include statement for the lazarus .lrs resource file is generated.

It should be obvious that the last class is suitable for registration of custom forms. To do this, one needs to implement a descendent of this file descriptor, which returns suitable information for the 'New' dialog:

1. Return a nice name to show in the 'New' dialog.
2. Return a nice description to show in the 'New' dialog.
3. Add the unit where the custom form is implemented to the source of the generated unit.

4. Return the correct class which the Lazarus IDE must create when editing an instance of this form class (Lazarus needs to retrieve the RTTI information of this class).

As this is rather cumbersome, a unit is implemented which takes care of the details. It exposes 3 calls:

```
Procedure RegisterCustomForm(AFormClass : TCustomFormClass;
                             Const AUnitName : String);
Procedure RegisterCustomForm(AFormClass : TCustomFormClass);
Procedure RegisterCustomForm(Descr : TCustomFormDescr);
```

The first of these calls needs only the form class pointer to register the form in the IDE, plus the name of the unit in which the class is implemented. The second call constructs the unit name by stripping the initial 'T' from the form class name.

The third call allows to specify more options. It accepts a small descriptor class as an argument:

```
TCustomFormDescr=Class
public
  Constructor Create(AFormClass : TCustomFormClass);
  Constructor Create(AFormClass : TCustomFormClass;
                     ACaption,ADescription,AUnit : String);
  Property FormClass : TCustomFormClass;
  Property Caption : String;
  Property Description : String;
  Property UnitName : String;
  Property Category : String;
  Property Author : String;
end;
```

The properties of this class are used to register the custom form class in the IDE. The caption is used in the 'New' dialog to represent the form class, the description and author are shown when this form is selected. The category is the category under which the item is shown in the 'New' dialog, by default this is 'Custom Forms'. The constructor of this class will fill in some default values for the variour properties, based on the `AFormClass` class name.

To register a form using this class, one would simply code:

```
RegisterCustomForm(TCustomFormDescr.Create(TMyForm,
                                           MyCaption,
                                           MyDescription,
                                           MyUnit));
```

Which is in fact what the other two overloaded versions of this call do:

```
Procedure RegisterCustomForm(AFormClass : TCustomFormClass);

begin
  RegisterCustomForm(TCustomFormDescr.Create(AFormClass));
end;

Procedure RegisterCustomForm(AFormClass : TCustomFormClass;
                             Const AUnitName : String);
Var
  D : TCustomFormDescr;

begin
  D:=TCustomFormDescr.Create(AFormClass);
  D.UnitName:=AUnitName;
  RegisterCustomForm(D);
end;
```

There is no need to free the description class, the custforms implementation will do this.

The main `RegisterCustomForm` call simply keeps the descriptions in a list:

```
Var
  CustomFormList : TObjectList;

Procedure RegisterCustomForm(Descr : TCustomFormDescr);

begin
  CustomFormList.Add(Descr);
end;
```

The list is created during the initialization of the unit, and freed when the unit is finalized (the description classes are freed with it).

Using the above description class, the custforms unit will register the custom forms in the lazarus IDE. For this, it implements a descendent of `TFileDescPascalUnitWithResource`, which looks as follows:

```
TCustomFormFileDescriptor = Class(TFileDescPascalUnitWithResource)
Public
  Constructor Create(ADescr : TCustomFormDescr);
  Property FormDescr : TCustomFormDescr Read FFormDescr;
  Function GetLocalizedName : String; override;
  Function GetLocalizedDescription : String; override;
  Function GetInterfaceUsesSection : String; override;
end;
```

The implementation of this IDE File description class is quite straightforward:

```
constructor TCustomFormFileDescriptor.Create(ADescr: TCustomFormDescr);
begin
  Inherited Create;
  FFormDescr:=ADescr;
  ResourceClass:=FFormDescr.FFormClass;
  Name:=FFormDescr.Caption;
end;
```

The form descriptor is saved, and the `ResourceClass` property is set. The `ResourceClass` property is the actual class that the Lazarus IDE will create when it needs to edit an instance of this class.

The other methods simply return the appropriate values from the custom form descriptor:

```
function TCustomFormFileDescriptor.GetLocalizedName: String;
begin
  Result:=FFormDescr.Caption;
end;

function TCustomFormFileDescriptor.GetLocalizedDescription: String;
begin
  Result:=FFormDescr.Description;
  If (FFormDescr.Author<>'') then
    Result:=Result+LineEnding+'By '+FFormDescr.Author;
end;
```

Last but not least, the unit in which the custom form class is implemented must be added to the uses clause of the unit:

```
function TCustomFormFileDescriptor.GetInterfaceUsesSection: String;
begin
  Result:=inherited GetInterfaceUsesSection;
  Result:=Result+',forms,'+FFormDescr.UnitName;
end;
```

Note that the 'Forms' unit is also added, as it contains the `TCustomForm` declaration.

Now that the descriptors are implemented and custom forms can be registered in `custforms`, all that needs to be done is to tell the Lazarus IDE about their existence. This is done in the 'Register' routine of the package:

```
Procedure Register;

Var
  L : TStringList;
  I : Integer;
  D : TCustomFormDescr;
  FD : TCustomFormFileDescriptor;

begin
  L:=TStringList.Create;
  Try
    L.Sorted:=True;
    L.Duplicates:=dupIgnore;
    For I:=0 to CustomFormList.Count-1 do
      L.Add(TCustomFormDescr(CustomFormList[i]).Category);
    For I:=0 to L.Count-1 do
      RegisterNewItemCategory(L[i]);
  Finally
    L.Free;
  end;
  For I:=0 to CustomFormList.Count-1 do
    begin
    D:=TCustomFormDescr(CustomFormList[i]);
    FD:=TCustomFormFileDescriptor.Create(D);
    RegisterProjectFileDescriptor(FD,D.Category);
    end;
end;
```

The first part of the routine collects all categories used by the custom form descriptors, and registers them in the IDE. The second part then creates a IDE file descriptor class for each registered custom form, and registers it with the IDE.

The custforms unit is inserted in the `lazcustforms` package, which must be installed in the IDE. By itself, this package does nothing. It must be used by another package to register new custom forms in the IDE.

To demonstrate this, a package with 2 custom form classes will be created. These forms will be registered using the custforms unit, and as such and added to the 'New' dialog in the lazarus IDE.

The two forms are descendent from `TCustomForm`, and they implement an extra form initialization method: `InitForm`. This initialization can be called at various times during the lifetime of the form. This is controlled by the `InitAt` property. It can have one of the following values:

ifaShow the initialization is called before the form is shown. Note that a form can be shown more than once.

ifaCreate the initialization is called after the form is created.

ifaActivate the initialization is called before the form is activated. Note that a form can be activated more than once.

The form that implements this is called `TAppForm`. The `InitForm` has 2 events associated with it: `BeforeInitForm` and `AfterInitForm`. It is implemented in the AppForm unit.

A descendent form of `TAppForm` is `TDBAppForm`. It overrides the `InitForm` method and in this methods, opens all datasets which are on the form. It introduces and additional boolean property `OpenDatasets` and 2 additional events `BeforeOpenDatasets` and `AfterOpenDatasets`. To control whether individual datasets should be opened, a `OnOpenDataset` event is introduced. It will be called before opening an individual dataset, and the user can prohibit the opening of the dataset. This form is implemented in the DBAppForm unit.

The code for these forms is actually quite simple:

```
procedure TAppForm.InitForm;
begin
  If Assigned(BeforeInitForm) then
    BeforeInitForm(Self);
  DoInitForm;
  If Assigned(AfterInitForm) then
    AfterInitForm(Self);
end;

procedure TAppForm.DoInitForm;
begin
  // Do nothing yet.
end;
```

The `DoInitForm` performs the actual work, for the demonstration it is left empty, in a real application it would be filled with initialization code. The `InitForm` method is static, but the `DoInitForm` is virtual, so it can be overridden in a descendent form, such as `TDBAppForm`.

Remains to call the `InitForm` at the currect time, based on the value of the `InitAt` propert:

```
constructor TAppForm.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  if (InitAt=ifaCreate) then
    InitForm;
end;

procedure TAppForm.DoShow;
begin
  If InitAt=ifaShow then
    InitForm;
  inherited DoShow;
end;

procedure TAppForm.Activate;
begin
  if (InitAt=ifaShow) then
    InitForm;
  inherited Activate;
end;
```

Note that the initialization must be called *after* the form was created: the property's value is read from the form file in the `TCustomForm.Create` method.

The `TDBAppForm` overrides the `DoInitForm` method to open all datasets on the form. This is implemented in a similar manner:

```
procedure TDBAppForm.DoInitForm;
begin
  inherited DoInitForm;
  If OpenDatasets then
    OpenAllDatasets;
end;

procedure TDBAppForm.OpenAllDatasets;
begin
  If Assigned(BeforeOpenDatasets) then
    BeforeOpenDatasets(Self);
  DoOpenDatasets;
  If Assigned(AfterOpenDatasets) then
    AfterOpenDatasets(Self);
end;
```

The `OpenAllDatasets` is a static method; the real work is again moved to a virtual method, which is a simple loop:

```
Procedure TDBAppForm.DoOpenDatasets;

Var
  I : Integer;
  D : TDataset;
  B : Boolean;

begin
  For I:=0 to ComponentCount-1 do
```

```
    begin
    If Components[i] is TDataset then
      begin
      D:=TDataset(Components[i]);
      B:=True;
      If Assigned(OnOpenDataset) then
        OnOpenDataset(D,B);
      If B then
        D.Open;
      end;
    end;
end;
```

The forms are registed in the IDE using a separate unit, regappforms:

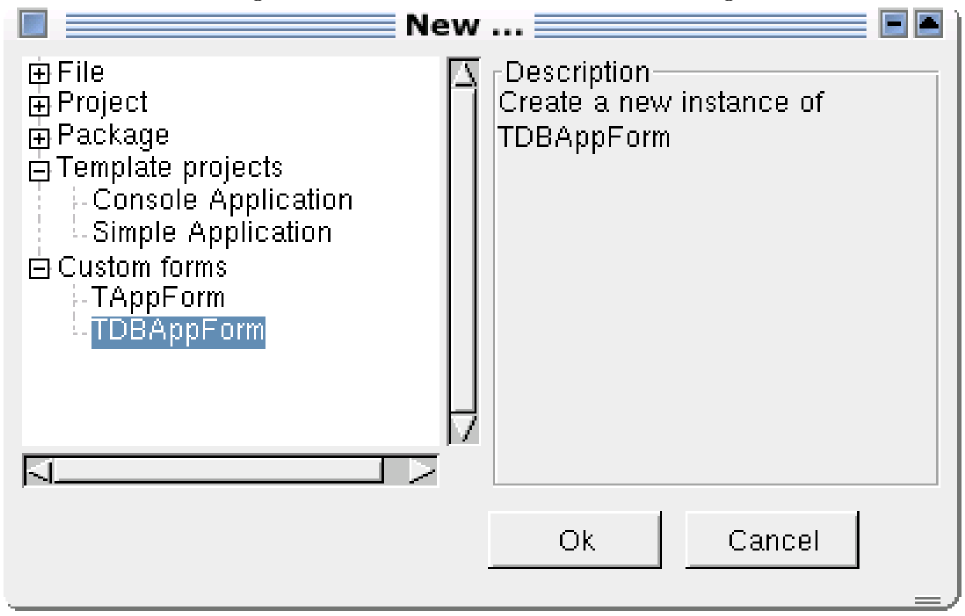
```
unit regappforms;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, appform, dbappform;

procedure RegisterAppForms;

implementation

uses custforms;

procedure RegisterAppForms;

begin
  RegisterCustomForm(TCustomFormDescr.Create(TAppForm));
  RegisterCustomForm(TCustomFormDescr.Create(TDBAppForm));
end;

initialization
  RegisterAppForms;
end.
```

There is a good reason for putting the registration code in a separate unit: if the forms were registered in the units where they are implemented, then the application (which uses the units) would also depend on the custforms unit, and hence on the Lazarus IDE interface units. The registration unit is normally not included in the end-user application, so the custforms unit is then also not included.

When the lazcustforms and appforms packages are installed in the IDE, then the new dialog will look something like in figure 4 on page 15.

When creating a new DBAppForm instance, it will be obvious that not all usual form properties will be present: only the properties that were published in the TAppForm and TDBAppform classes are shown in the object inspector, just as for the events. This is shown in figure 5 on page 16.

Figure 4: Custom forms in the 'New' dialog



Create a new instance of TDBAppForm

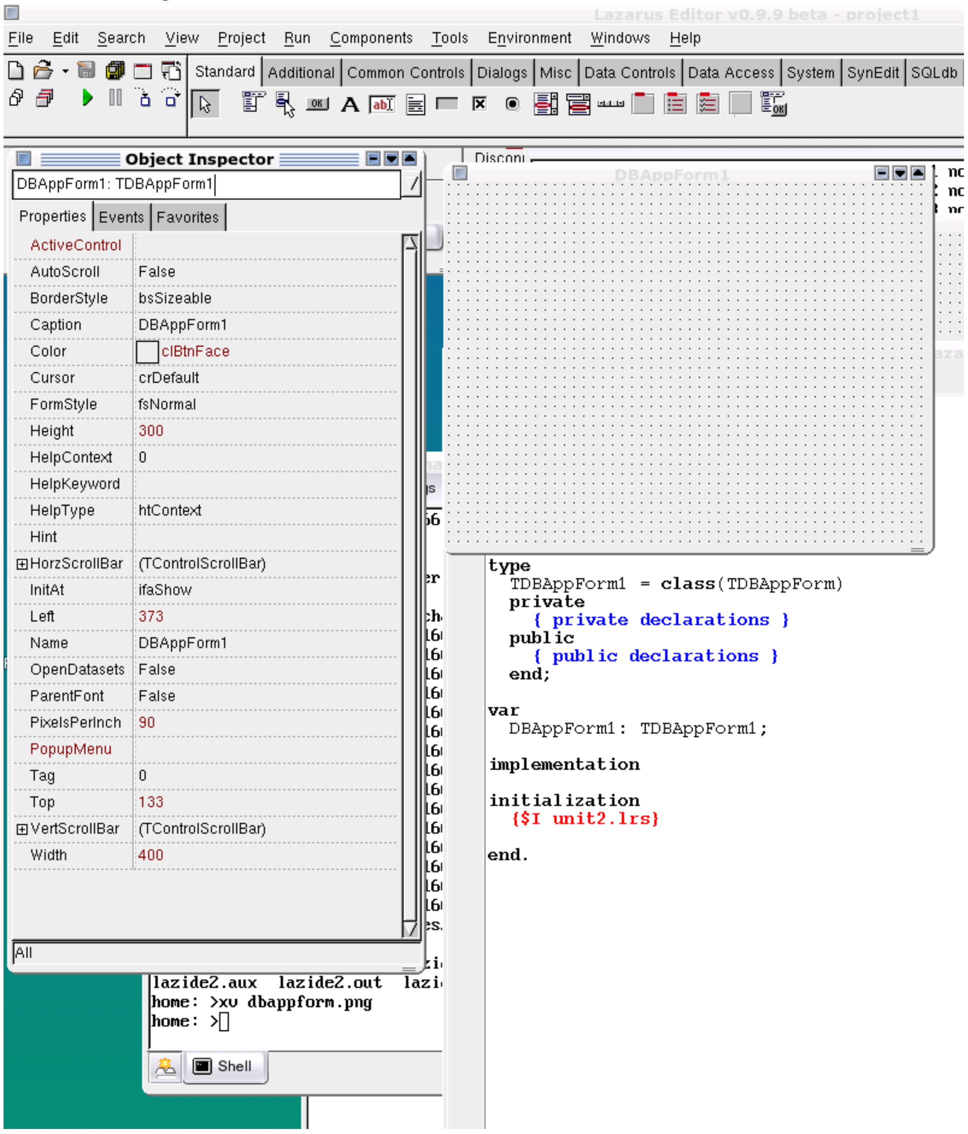
6 Conclusion

Extending the Lazarus IDE is not entirely trivial, but it is not very difficult either; In fact, the author thinks it's easier to do in the Lazarus IDE than in Delphi. The fact that the Lazarus IDE sources are available can only be considered a plus.

In this and the previous article, 2 packages were developed: One package to allow easy addition of complete standard projects, based on a project template, and a package to add custom form classes to the Lazarus IDE. These packages not only serve to illustrate the concepts of the Lazarus IDE, they can be put to good use as well. In fact, they are already put to use.

This is not yet the end of the Lazarus IDE interface's possibilities. The IDE can be extended in other ways as well, for example by adding menu items to the lazarus IDE. But this is left to a future contribution.

Figure 5: A TDBAppForm instance in the Object Inspector


Lazarus Editor v0.9.9 beta - project1
File Edit Search View Project Run Components Tools Environment Windows Help
Standard Additional Common Controls Dialogs Misc Data Controls Data Access System SynEdit SQLdb
Object Inspector
DBAppForm1: TDBAppForm1
Properties Events Favorites
All
DBAppForm1
Shell

ActiveControl	
AutoScroll	False
BorderStyle	bsSizeable
Caption	DBAppForm1
Color	⬜ clBtnFace
Cursor	crDefault
FormStyle	fsNormal
Height	300
HelpContext	0
HelpKeyword	
HelpType	htContext
Hint	
⊞HorzScrollBar	(TControlScrollBar)
InitAt	ifaShow
Left	373
Name	DBAppForm1
OpenDatasets	False
ParentFont	False
PixelsPerInch	90
PopupMenu	
Tag	0
Top	133
⊞VertScrollBar	(TControlScrollBar)
Width	400

```
type
  TDBAppForm1 = class(TDBAppForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  DBAppForm1: TDBAppForm1;

implementation

initialization
  {$I unit2.lrs}

end.
```

```
lazide2.aux  lazide2.out  lazi
home: >xv dbappform.png
home: >
```