Extending the Lazarus IDE: Custom forms and units

Michaël Van Canneyt

1st June 2005

Abstract

To extend the Lazarus IDE, it is not necessary to edit the Lazarus sources. The Lazarus IDE is extensible with custom packages, and offers an API to integrate a package in the IDE. In this article, we'll examine one of the ways in which the Lazarus IDE can be extended, namely by adding items to the "New" dialog.

1 Introduction

The Lazarus IDE is open source, so in principle, anyone can take the IDE sources, and add custom extensions to the IDE. This is cumbersome, and is hard to maintain. Fortunately, the Lazarus IDE interface offers a way to extend the functionality of the IDE. All that needs to be done is to create a package which is compiled and installed in the IDE, after which the functionality offered by the package will be available. This is no different from the way Delphi does it.

To demonstrate this, a package will be made which hooks into the File-New dialog in the IDE. It allows to create new projects, based on project templates, located in some directory. On the basis of the files in the template directory, a new project is created, and filled with files. The templates can contain variables, which will be substituted with values that the user can supply, such as the project name, a name for the main form etc..

2 Template organisation

The template repository is a simple directory, with a subdirectory for each template. The subdirectory may contain other directories, which will be copied as part of the project creation process. In the directory for each template, 2 files should exist:

project.ini contains the name of the project, plus definitions of the variables used in the template.

description.txt a file containing a more verbose description of the project. This description (a simple text) will be shown in the IDE when the project is selected.

The project.ini file is in windows .INI format. It contains 2 sections. The first section is called `Global` and can contain the following key=value pairs:

Name The name of the project.

Author The author of the project.

Description a one-line description of the project.

Recurse a boolean (0 or 1) which indicates whether subdirectories are part of the templare or not.

Exclude a comma separated list of extensions to be excluded from variable substitution (for instance binary files). These files will simply be copied.

The second section is called `Variables` and should contain a `key=Value` pair for each variable that should be substituted when creating the project files. The value should be a one-line description, which will be shown in the IDE.

The following is a sample project.ini file:

```
[project]
Name=Console Application
Author=Michael Van Canneyt

[variables]
ProjName=Name of the project file.
MyMainFormName=Name for the main form of the application.
AppName=Name of the application class.
```

The `ProjName` variable does not need to be defined here. If it is encountered, it will automatically be substituted with the project name.

Variables should be specified in a format similar to the GNU Makefile variables:

```
T$(MyMainFormName) = Class(TForm)
...
end;
```

Here `MyMainFormName` is the name of the variable.

Note that not only the sources may contain variables. The filenames themselves can also contain variables, they will be substituted as well.

The IDE package will be split in 2 main parts:

1. A class to manage templates, and to create projects or files from templates. It takes care of copying files, and performs the necessary variable name substitution.
2. A class which registers the templates in the IDE.

The first class is implemented as a descendent of `TCollection`, the second class is a `TProjectDescriptor` decsendent. The `TProjectDescriptor` is a class, defined by the IDE interface.

3 Managing the project templates

To manage the templates, the `TProjectTemplates` class is defined. It is a descendent of `TCollection` with a quite simple interface:

```
Constructor Create(Const ATemplateDir : String);
Procedure Initialize(Const ATemplateDir : String);
Procedure CreateProject(Const ProjectName, ProjectDir: String;
```

```
                          Variables : TStrings);
Function IndexOfProject(Const ProjectName : String) : Integer;
Function ProjectTemplateByName(
  Const ProjectName : String) : TProjectTemplate;
Property TemplateDir : String;
Property Names [Index : Integer] : String;
Property Templates[Index : Integer] : TProjectTemplate;default;
```

The meaning of these methods should be obvious from their names:

Create creates a new template collection, and initializes it from `TemplateDir`.

Initialize Initializes the templates from directory `ATemplateDir`. This can be used to change the template directory.

IndexOfProject returns the index of a template, based on it's name. returns -1 if the project was not found.

ProjectTemplateByName returns the `TProjectTemplate` instance based on the name.

CreateProject will create a project, based on the template with name `TemplateName`. The project will be created in directory `ProjectDir`, and any variables will be substituted from the `Name=Value` pairs in the `Variables` stringlist.

Names is an indexed property containing the names of the found templates.

Templates is the default property of the `TProjectTemplates` class, providing indexed access to the individual `TProjectTemplate` instances.

The interesting methods in this class are `Initialize` and `CreateProject`. The initialize method initializes the template definitions. It simply looks for the subdirectories in the given template directory, and initializes a `TProjectTemplate` instance from the found subdirectories:

```
procedure Initialize(const ATemplateDir: String);

Var
  Info : TSearchRec;
  D : String;

begin
  Clear;
  FTemplateDir:=IncludeTrailingPathDelimiter(ATemplateDir);
  D:=FTemplateDir;
  If FindFirst(D+'*',faDirectory,Info)=0 then
    try
      Repeat
        If ((Info.Attr and faDirectory)<>0)
           and not ((Info.Name='.') or (Info.Name='..')) then
          With Add as TProjectTemplate do
            InitFromDir(D+Info.Name);
      Until FindNext(Info)<>0;
    finally
      FindClose(Info);
    end;
end;
```

This method could be expanded with code to ignore certain directories, for instance CVS or `.svn` directories.

The `CreateProject` method passes the actual work to the needed `TProjectTemplate` instance:

```
procedure CreateProject(const ProjectName, ProjectDir: String;
                        Variables : Tstrings);

Var
  T : TProjectTemplate;

begin
  T:=ProjectTemplateByName(ProjectName);
  T.CreateProject(ProjectDir,Variables);
end;
```

The `CreateProject` method of `TProjectTemplate` is presented below.

The `TProjectTemplate` class is a descendent of `TCollectionItem`, and has the following properties:

```
Property Name : String;
Property Directory : String;
Property Description : String;
Property Variables : TStrings;
Property Author : String;
Property Recurse : Boolean;
Property Exclude : String;
Property FileCount : Integer;
Property Files[Index : Integer] : String;
```

The properties correspond to the values found in the `project.ini` file, except for `FileCount`, which is the number of files found in the template, and the `Files` property, which gives the names of all files found in the template. The `Variables` contain the variables as defined in the project file, together with their descriptions, not the values that will be substituted.

The following public methods exist:

```
Procedure CreateProject(Const ProjectDir : String; Values : TStrings);
Procedure CreateFile(FileIndex : Integer; Source,Values : TStrings);
Procedure CreateFile(Const FileName: String; Source,Values : TStrings);
Procedure CreateProjectDirs(Const BaseDir : String; Values : TStrings);
Function TargetFileName(FN : String; Values : TStrings) : String;
Function TargetFileName(I : Integer; Values : TStrings) : String;
```

The meaning of these methods should be obvious:

CreateProject creates a project based on the template files, in directory `ProjectDir`. The `Values` stringlist contains the values to be used when substituting variable.

CreateFile Creates a file from source file specified by index or filename. The contents of the file, with variables expanded, is returned in `Source`. The values for substitution are taken from `Values`.

CreateProjectDirs recreates, under `BaseDir`, the directory tree found in the project template directory, expanding variable names with the values found in `Values`.

TargetFileName returs a filename, relative to the project directory, for a source filename specified by index or name. It replaces any variables found in the filename with values found in the `Values` stringlist.

The `InitFromDir` method a private method is used to initialize the template. It reads the template settings from the `project.ini` file, and retrieves the list of files in the project:

```
procedure TProjectTemplate.InitFromDir(const DirName: String);

Var
  L : TStringList;
  FN : String;

begin
  FDirectory:=IncludeTrailingPathDelimiter(DirName);
  L:=TStringList.Create;
  Try
    FN:=FDirectory+'project.ini';
    If FileExists(FN) then
      begin
      With TMemInifile.Create(FN) do
        try
          FName:=ReadString(SProject,KeyName,DirName);
          FAuthor:=ReadString(SProject,KeyAuthor,'');
          FDescription:=ReadString(SProject,KeyDescription,'');
          FRecurse:=ReadBool(SProject,KeyRecurse,False);
          FExclude:=ReadString(SProject,KeyExclude,'');
          If (FExclude<>'') then
            FExclude:=FExclude+',';
          ReadSectionValues(SVariables,FVariables);
        Finally
          Free;
        end;
      end;
    FN:=Directory+'description.txt';
    If FileExists(FN) then
      begin
      L.LoadFromFile(FN);
      FDescription:=L.Text;
      end;
    GetFileList(FDirectory);
  Finally
    L.Free;
  end;
end;
```

The main method is `CreateProject`:

```
procedure CreateProject(const ProjectDir: String;
                        Values: TStrings);

begin
  CopyAndSubstituteDir(Directory,ProjectDir,Values);
end;
```

It leaves the real work to `CopyAndSubstitureDir`, which is a recursive method. It first copies all files in the the source directory to the target directory, expanding variables as it goes. Then it calls itself for each subdirectory found in the source directory (if recursion was not disabled):

```
procedure CopyAndSubstituteDir(Const SrcDir,DestDir :String;
                               Values: Tstrings);

Var
  D1,D2 : String;
  Info : TSearchRec;

begin
  D1:=IncludeTrailingPathDelimiter(SrcDir);
  D2:=IncludeTrailingPathDelimiter(DestDir);
  If not ForceDirectories(D2) then
    Raise ETemplateError.CreateFmt(SErrCouldNotCreateDir,[D2]);
  If FindFirst(D1+'*',0,Info)=0 then
    try
      repeat
        if (info.name<>'description.txt')
           and (info.name<>'project.ini') then
          CopyAndSubstituteFile(D1+Info.Name,
                                D2+SubstituteString(Info.Name,
                                                    Values),
                                Values);
       Until (FindNext(Info)<>0);
    finally
      FindClose(Info);
    end;
  if Recurse then
    If (FindFirst(D1+'*',0,Info)<>0) then
      try
        repeat
          if ((Info.attr and faDirectory)<>0) and
            (Info.Name<>'.') and (info.Name<>'..') then
            CopyAndSubstituteDir(D1+Info.Name,
                                 D2+SubstituteString(Info.Name,
                                                     Values),
                                 Values);
        until FindNext(Info)<>0;
      finally
        FindClose(Info);
      end;
end;
```

Note that variables in the names of the target files or directories are substituted with the value of the variables using the `SubstituteString` function.

Copying a file is done in the `CopyAndSubstituteFile` method:

```
procedure TProjectTemplate.CopyAndSubstituteFile(Const SrcFN,DestFN :
String; Values : Tstrings);

Var
```

```
  L : TStrings;

begin
  If pos(ExtractFileExt(SrcFN)+',',Exclude)<>0 then
    begin
    If not SimpleFileCopy(SrcFN,DestFN) then
      Raise ETemplateError.CreateFmt(SErrFailedToCopyFile,[SrcFN,DestFN]);
    end
  else
    begin
    L:=TstringList.Create;
    try
      CreateFile(SrcFN,L,Values);
      L.SaveToFile(DestFN);
    Finally
      L.Free;
    end;
    end;
end;
```

This method checks whether the file must be copied as-is (which is done in SimpleFileCopy), or whether the contents of the file must be checked for variables, in which case the public `CreateFile` method is used. This method is quite easy:

```
procedure TProjectTemplate.CreateFile(const FileName: String; Source,
  Values: TStrings);

Var
  F : Text;
  Line : String;

begin
  AssignFile(F,FileName);
  Reset(F);
  Try
    While not EOF(F) do
      begin
      ReadLn(F,Line);
      Source.Add(SubstituteString(Line,Values));
      end;
  Finally
    CloseFile(F);
  end;
end;
```

This covers the most important methods in the `TProjectTemplate` class. The two classes described here are implemented in the projecttemplates unit. This unit is independent of the Lazarus IDE, and could be used in any project. In the next section, we explain how to use this unit in the Lazarus IDE.

4 The various Lazarus IDE interfaces

In Lazarus, the equivalent of the Delphi 'Open Tools API' is the 'Lazarus IDE interface'. This is a collection of units which expose various elements of the Lazarus IDE so they can be used in packages. The interface consists of a set of base classes, together with some global variables which are set by the IDE. These units are located in the ideintf directory in the Lazarus source directory. This directory contains the following units:

LazIDEIntf Contains a general interface to the IDE, which allows to open files, and retrieve some configuration information.

NewItemIntf Contains an interface to the **'File|New'** menu dialog. It allows to add item categories, such as the pre-defined 'File', 'Project' or 'Package'.

ProjectIntf Contains an interface to create new items in the 'File' and 'Project' category of the **'File|New'** dialog. This is the unit which will be used below.

HelpIntf An interface to the help system of lazarus. The help system is very extensible.

ConfigStorage An interface to the lazarus configuration system. This can be used if an IDE package needs to save/restore settings. The settings will be stored in the Lazarus settings directory.

FormEditingIntf Defines interfaces to form and component editors.

PackageIntf An interface to the IDE package system. It can be used to introduce new packages (in the 'Package' category) in the **'File|New'** dialog.

MacroIntf An interface to the IDE Macros interface. The macros are used in the tools, build commands, configuration settings. However, it does not include the possibility to define additional macros, so it was not used to implement the project templates.

IDECommands An interface to IDE commands. This can be used to add command keystrokes to the IDE.

ActionsEditor an interface to implement and register custom `TAction` descendents, which works much as in Delphi.

ComponentEditor an interface to component editors, compatible to the Delphi version.

PropEdits an interface to property editors, compatible to the Delphi version.

ObjectInspector Contains the Object Inspector class. It simply manages the various property editors.

There are other units in this directory, with some examples of component editors.

5 The project interface

Of all the interfaces presented in the previous section, only the Lazarus IDE interface, configuration storage, and the project interface will be used.

The project interface is defined in the ProjectIntf unit. It defines a number of classes which can be used to extend the IDE with custom projects and files. The IDE itself uses the classes in this interface to implement the standard classes.

The two most important classes in this unit are the following

TProjectDescriptor A class to implement a new project in the IDE. When the IDE needs to create a new project, it does so based on the properties and methods found in this class.

TProjectFileDescriptor A class to implement a new file (a unit or program file) in the IDE. When the Lazarus IDE needs to create a new source file, it uses the methods and properties if this class to do so.

How should these classes be used ? Simple: A descendent of these classes must be made, and it should either set some properties, or override some methods so the Lazarus IDE can use it to create a new project or file.

An instance of this descendent class must be registered in the Lazarus IDE, so the IDE is aware of the new project type. When the IDE needs to create a new file of the registered type, the methods of the instance will be used, as will be demonstrated below.

The first of the two classes that is needed to implement the template projects is the `TProjectDescriptor` class. It has the following (simplified) declaration:

```
TProjectDescriptor=class(TPersistent)
function DoInitDescriptor: TModalResult; virtual;
function GetLocalizedName: string; virtual;
function GetLocalizedDescription: string; virtual;
function InitProject(AProject: TLazProject): TModalResult;
                                              virtual;
function CreateStartFiles(AProject: TLazProject): TModalResult;
                                                   virtual;
property Name: string;
property VisibleInNewDialog;
property Flags: TProjectFlags;
property DefaultExt;
end;
```

The `Name` property identifies the kind of project in the IDE, but is otherwise not used. The `VisibleInNewDialog` property (standard `True`) determines whether a project of this type will be shown in the 'New' dialog. If so, it will be shown with a caption as returned by `GetLocalizedName`, and when it is selected, the description as returned by `GetLocalizedDescription` will be shown.

When the user selects the kind of project as described by the descriptor, the following happens:

1. IDE will call the `DoInitDescriptor` method. This method should be used to show a dialog which can be used to ask some additional information of the user.
2. If the `DoInitDescriptor` method returns `mrOK` (the default behaviour), then the IDE will discard the currently active project, and starts a new project.
3. After some internal initialization, `InitProject` is called with a new `TLazProject` instance, which represents the new project. The `TProjectDescriptor` can set some properties of `TLazProject` (e.g. compiler options) or call its methods. If this method returns a value other than `mrOK`, the new project is abandoned.
4. After the `InitProject` has finished with `mrOk`, the `CreateStartFiles` routine is called. This method should also return `mrOK`, and should be used to add files to the project.

After these methods are called, the new project is ready for use in the IDE. If `CreateStartFiles` was used to create some files, then they can be opened in the IDE, or they may already be open, depending on the options used.

The `TLazProject` class that is passed to the last two methods, has the following interface declaration:

```
function CreateProjectFile(
         const Filename: string): TLazProjectFile;
procedure AddFile(ProjectFile: TLazProjectFile;
                  AddToProjectUsesClause: boolean);
procedure RemoveUnit(Index: integer);
procedure AddSrcPath(const SrcPathAddition: string);
procedure AddPackageDependency(const PackageName: string);
property MainFileID: Integer;
property Files[Index: integer]: TLazProjectFile;
property FileCount: integer read GetFileCount;
property MainFile: TLazProjectFile;
Property Title: String;
property Flags: TProjectFlags read FFlags write SetFlags;
property LazCompilerOptions: TLazCompilerOptions;
property ProjectInfoFile: string;
```

Of these, `CreateProjectFile` must be called in order to create the program .lpr file. The Project info file (the .lpi) file name can be set using the `ProjectinfoFile` property. The `Title` property can be used to set the title of the application. The `Files` property provides indexed access to all files defined within the project (as shown in the project inspector), and in this indexed list, the file with index `MainFileID` is the main project file (the program file). The `Flags` property contains a set of the following flags which control the behaviour of the IDE, and how it treats the main source file for the project:

pfSaveClosedUnits Tells the IDE to save information about closed files (which are not part of the project) in the project information file.

pfSaveOnlyProjectUnits Tells the IDE not to save information about foreign files

pfMainUnitIsPascalSource Tells the IDE that the main unit is a pascal source file even if the extension is not .pas or .pp.

pfMainUnitHasUsesSectionForAllUnits Tells the IDE to add/remove all pascal units to the main file's uses section.

pfMainUnitHasCreateFormStatements Tells the IDE to add/remove `Application.CreateForm` statements to the main file for each form that must be auto-created.

pfMainUnitHasTitleStatement Tells the IDE to add/remove a `Application.Title:=` statement to the main source file.

pfRunnable tells the IDE that this project can be run.

`LazCompilerOptions` is a reference to the compiler settings used when compiling this project. The declaration of this class is very long, the reader is referred to the source of the projectintf unit for details.

The `CreateProjectFile` must be used to create the main project file, as this is a project file, not a unit. How the project file's source is created depends on the `Flags` property. Normal files/units can be added to the project with the `AddFile` call. They can

be removed with the `RemoveUnit` path. The dependency of the project on other packages, such as the LCL or FCL, can be indicated using the `AddPackageDependency` call.

Armed with these classes, the template project descriptor can be declared as follows:

```
TTemplateProjectDescriptor = class(TProjectDescriptor)
Private
  FTemplates : TProjectTemplates;
  FTemplate : TProjectTemplate;
  FProjectDirectory : String;
  FProjectName : String;
  FVariables : TStrings;
protected
  Procedure InitTemplates;
  procedure SaveTemplateSettings;
public
  constructor Create;
  destructor destroy;
  Function DoInitDescriptor : TModalResult;
  function GetLocalizedName: string;
  function GetLocalizedDescription: string;
  function InitProject(AProject: TLazProject) : TModalResult;
  function CreateStartFiles(AProject: TLazProject) : TModalResult;
end;
```

The `FTemplates` field will keep a reference to the `TProjectTemplates` defined earlier. The `FTemplate` variable will be used to keep the template chosen by the user, just as the `ProjectDirectory`, `ProjectName` and `FVariables` fields.

A single instance of this class is instantiated in the `Register` procedure of the package in which the templates are implemented:

```
Var
  TemplateProjectDescriptor : TTemplateProjectDescriptor;

procedure Register;
begin
  TemplateProjectDescriptor:=TTemplateProjectDescriptor.Create;
  RegisterProjectDescriptor(TemplateProjectDescriptor);
end;
```

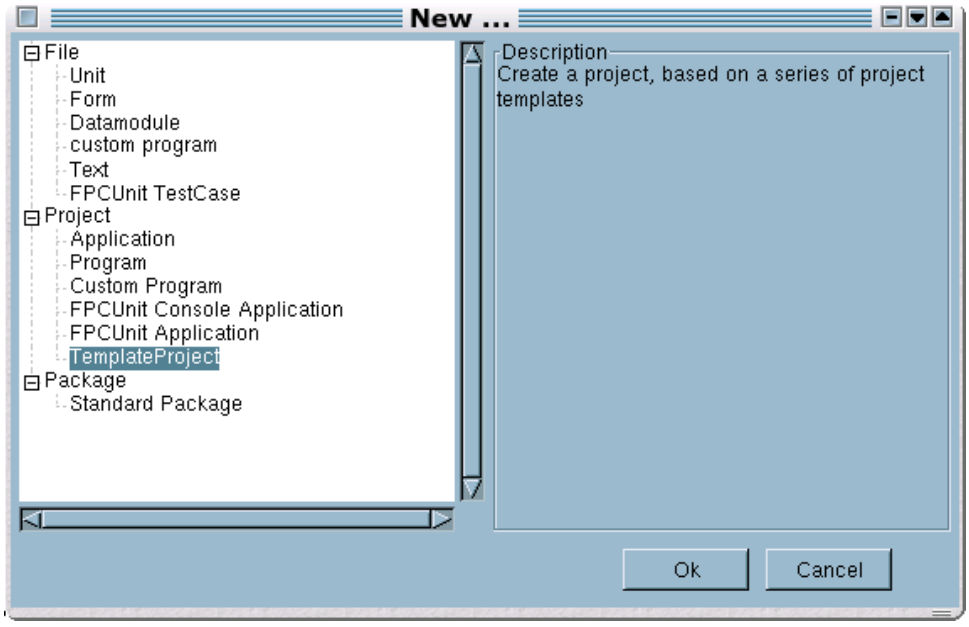
The `RegisterProjectDescriptor` call is defined in the projectintf unit. This means that the project descriptor 'lives' as long as the package is loaded, i.e., currently from the start of the IDE till it is closed.

The functions `GetLocalizedName` and `GetLocalizedDescription` determine what the IDE shows to the user in the 'New' dialog:

```
function TTemplateProjectDescriptor.GetLocalizedName: string;
begin
  Result:='Template Project';
end;

function TTemplateProjectDescriptor.GetLocalizedDescription: string;
begin
  Result:='Create a project, based on a series of project templates';
end;
```

Figure 1: The entry in the 'New' dialog



The result is shown in figure 1 on page 12.

The constructor and destructor are quite straightforward:

```
constructor TTemplateProjectDescriptor.Create;
begin
  inherited Create;
  Name:='TemplateProject';
  FVariables:=TStringList.Create;
end;

destructor TTemplateProjectDescriptor.destroy;
begin
  FTemplate:=NIl;
  FreeAndNil(FTemplates);
  FreeAndNil(FVariables);
  Inherited;
end;
```

Note that the `FTemplates` is not initialized in the constructor. To save memory, initialization is only done when the project descriptor is activated for the first time. Then, a call to `InitTemplates` is made:

```
procedure TTemplateProjectDescriptor.InitTemplates;

Var
  D,P : String;

begin
  If (FTemplates=Nil) then
    With GetIDEConfigStorage('projtemplate.xml',True) do
```

```
      try
        P:=LazarusIDE.GetPrimaryConfigPath;
        P:=IncludeTrailingPathDelimiter(P)+'templates';
        D:=GetValue('TemplateDir',P);
        FTemplates:=TProjectTemplates.Create(D);
      Finally
        Free;
      end;
end;
```

The `GetIDEConfigstorage` call is part of the `ConfigStorage` interface to the IDE. It manages all configuration files: it creates configuration files in a centralized location. The return value of this function is an instance of `TConfigStorage`, which is much like an ini-file or registry, only that the settings file has an XML format. The `True` parameter to this call indicates that the stored info should be read from disk. The `GetValue` call is part of the `TConfigStorage` class:

```
function  GetValue(APath, ADefault: String): String;
function  GetValue(APath: String; ADefault: Integer): Integer;
function  GetValue(APath: String; ADefault: Boolean): Boolean;
```

it reads a value (string, integer, boolean) from the XML file. The location in the XML file is determined by `APath`, which follows an XPATH-like syntax. In the code above, the `GetValue` call is used to read the location of the template directory. Based on this value, the templates are initialized.

The value of `P` is used as a default value. It is used by the `TConfigStorage` if no appropriate value was found in the configuration file. The value of `P` is obtained from the Lazarus IDE interface: the `GetPrimaryConfigPath` call returns the location of the Lazarus configuration files (more on this follows below). The templates are assumed to be in a subdirectory of this location.

The `InitTemplates` procedure is called when the Lazarus IDE calls the `InitDescriptor`, i.e. when the user has selected the project type corresponding to the project descriptor in the 'New' dialog:

```
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;
    end;
```

After initializing the templates, the options dialog is shown: this allows the user to select which template should be used, what the name of the project will be, and where the project should be located, as can be seen in figure figure 2 on page 14. If the user closed the dialog succesfully, and there are variables for which a value must be asked, the variable dialog is shown (figure figure 3 on page 14). If the user closed this dialog with succes, then the standard `ProjName` and `ProjDir` variables are defined.

The `ShowOptionsDialog` function is quite simple, really:

Figure 2: The options dialog: selecting a template

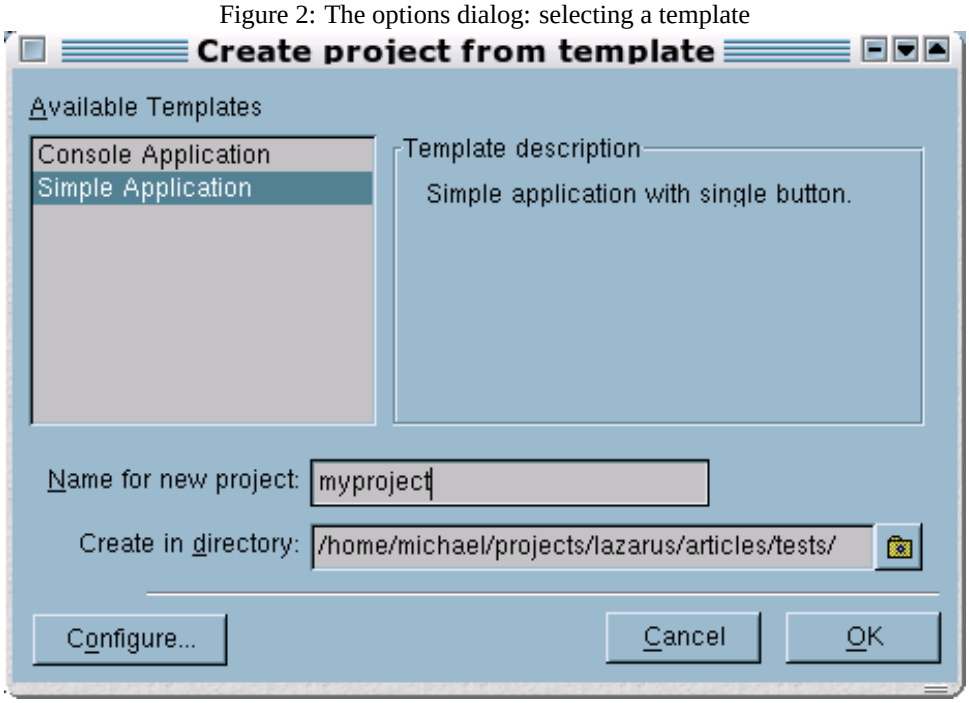
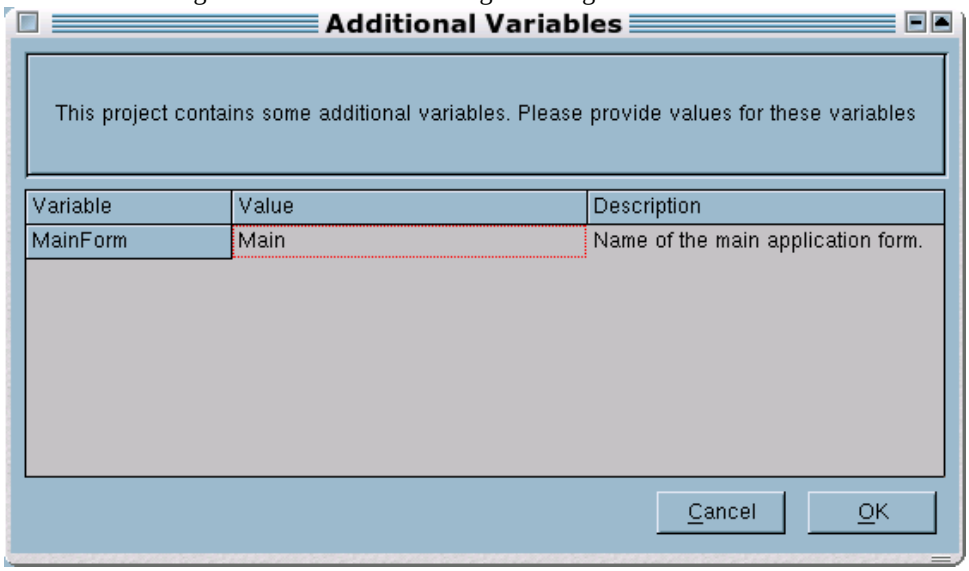


Figure 3: The variables dialog: defining additional variabls



```
function TTemplateProjectDescriptor.ShowOptionsDialog : TModalResult;

var
  I: Integer;

begin
  with TTemplateOptionsForm.Create(Application) do
    try
      Templates:=Self.FTemplates;
      Result:=ShowModal;
      if Result=mrOK then
        begin
        FProjectDirectory:=
          IncludeTrailingPathDelimiter(ProjectDir);
        FProjectName:=ProjectName;
        FTemplate:=Template;
        FVariables.Assign(FTemplate.Variables);
        I:=FVariables.IndexOfName('ProjName');
        if (I<>-1) then FVariables.Delete(I);
        I:=FVariables.IndexOfName('ProjDir');
        if (I<>-1) then FVariables.Delete(I);
        end;
      if SettingsChanged then
        SaveTemplateSettings;
    finally
      Free;
    end;
end;
```

If by any chance, the `ProjName` and `ProDir` are in the list of variables, they are deleted, so they will not be shown to the user when values are asked for other variables. If the user pressed the configure button in the dialog (indicated by the `SettingsChanged` property), then the new settings are saved in the `SaveTemplateSettings` method:

```
procedure TTemplateProjectDescriptor.SaveTemplateSettings;

begin
  With GetIDEConfigStorage('projtemplate.xml',False) do
    try
      SetValue('TemplateDir',FTemplates.TemplateDir);
      WriteToDisk;
    Finally
      Free;
    end;
end;
```

This method does essentially the opposite of the `InitTemplates` method discussed above. Note that the `GetIDEConfigStorage` gets a second parameter which indicates that it is only needed for writing all settings; any previous information in the file will be lost.

After all this, the `InitDescriptor` call of the project descriptor has finished. If it returned `mrOK`, the IDE will now close any open project and start a new project. it will then call the `InitProject` method:

```
function InitProject(AProject: TLazProject) : TModalResult;

Var
  I : Integer;
  AFile: TLazProjectFile;
  FN : String;
  B : Boolean;
  RFN : String;
  L : TStringList;

begin
  AProject.AddPackageDependency('FCL');
  AProject.AddPackageDependency('LCL');
  AProject.Title:=FProjectName;
  FTemplate.CreateProjectDirs(FProjectDirectory,FVariables);
  AProject.ProjectInfoFile:=FProjectDirectory
                            +FProjectName+'.lpi';
  For I:=0 to FTemplate.FileCount-1 do
    begin
    FN:=FTemplate.FileNames[I];
    B:=CompareText(ExtractFileExt(FN),'.lpr')=0;
    If B then
      begin
      FN:=FProjectDirectory+
          FTemplate.TargetFileName(FN,FVariables);
      AFile:=AProject.CreateProjectFile(FN);
      AFile.IsPartOfProject:=true;
      AProject.AddFile(AFile,Not B);
      AProject.MainFileID:=0;
      L:=TstringList.Create;
      try
        FTemplate.CreateFile(I,L,FVariables);
        AFile.SetSourceText(L.Text);
      Finally
        L.Free;
      end;
      end;
    end;
  Result:=mrOK;
end;
```

This function starts by adding dependencies on the FCL and LCL packages to the project, and sets the project title. Based on the settings supplied by the user, It then proceeds by creating all directories in that exist in the template project at the new project location. It sets the name of the project information file. It then searches for the project file in the template, and creates a file desciptor for this via the `CreateProjectFile` method of `TLazProject`. This returns a `TLazProjectFile` instance, which is added to the project as the main project file. Last but not least, the source for the project file is loaded from the template, and set via the `SetSourceText` method of the `TLazProjectFile` instance.

At this point, the Lazarus IDE has started the project, and has added the project source file to it. Now the rest of the project files must still be created. This is done in the `CreateStartFiles` function, which is called next by the IDE:

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

Const
  IDeOpts=[nfIsPartOfProject,nfOpenInEditor,nfCreateDefaultSrc];

Var
  Descr : TProjectFileDesc;
  I : Integer;
  FN,FN2 : String;
  B : Boolean;

begin
  Descr:=TProjectFileDesc.Create(FTemplate,FVariables);
  Try
    For I:=0 to FTemplate.FileCount-1 do
      begin
      FN:=FTemplate.FileNames[I];
      B:=CompareText(ExtractFileExt(FN),'.lpr')<>0;
      If B then
        begin
        B:=CompareText(ExtractFileExt(FN),'.lfm')<>0;
        If B then
          begin
          FN2:=ChangeFileExt(FN,'.lfm');
          B:=FileExists(FN2);
          FN:=FProjectDirectory+
              FTemplate.TargetFileName(FN,FVariables);
          If B then
            Descr.ResourceClass:=TForm;
          Descr.FIndex:=I;
          LazarusIDE.DoNewEditorFile(Descr,FN,'',IdeOpts)
          end;
        end;
      end;
  Finally
    Descr.Free;
  end;
  Result:=mrOK;
end;
```

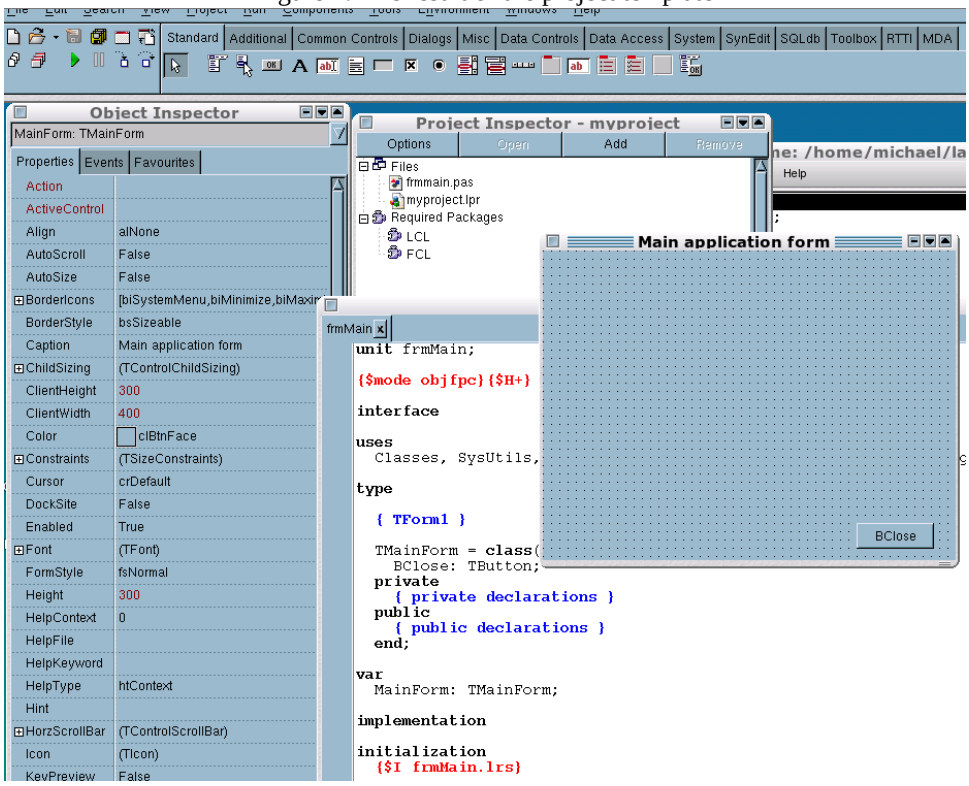
The method starts by creating an instance of the `TProjectFileDesc` class. This class is used by Lazarus to describe a new file for a project, and will be discussed in more detail below. Then the procedure loops over all files in the template; it skips the project file and the form files. For each other file it saves the index in the `TProjectFileDesc` instance, and sets the `ResourceClass` field to `TForm` if it finds a form file next to the unit file. This will tell lazarus what class it should create in order to display the 'form' (a module, a form etc.).

Then it calls the Lazarus IDE interface:

```
LazarusIDE.DoNewEditorFile(Descr,FN,'',IdeOpts)
```

This tells the Lazarus IDE to create a new file, based on the source file descriptor `Descr`, with filename `FN` and empty source. The options passed in `IdeOpts` tell the IDE that the file should be part of the project, that the file should be opened in the editor, and that it

Figure 4: The result of the project template



should create a default source for the file.

The actual work of creating the source file is then handled by the `TProjectFileDesc` class, discussed in the next section.

At this point, the Lazarus IDE has finished creating the new project. It will look as in figure 4 on page 18.

6 The file interface

In the previous section, the `TProjectFileDesc` class was used to create the source files for the project. This class is a descendent of `TProjectFileDescriptor`. This class is used by the IDE when it needs to create a new file; This can be a unit, a form, any kind of file. The IDE uses this class internally also to create the file items found in the IDE 'New' dialog. The class has the following public interface:

```
function GetLocalizedName: string; virtual;
function GetLocalizedDescription: string; virtual;
function GetResourceSource: string; virtual;
function CreateSource(const Filename, SourceName,
                      ResourceName: string): string; virtual;
property Name: string;
property DefaultFilename: string;
property DefaultFileExt: string;
property DefaultSourceName: string;
property DefaultResFileExt: string;
```

```
property DefaultResourceName: string;
property RequiredPackages: string;
property ResourceClass: TPersistentClass;
property IsComponent: boolean read FIsComponent;
property UseCreateFormStatements: boolean;
property VisibleInNewDialog: boolean
property IsPascalUnit: boolean;
property AddToProject: boolean;
```

The `VisibleInNewDialog`, `GetLocalizedName` and `GetLocalizedDescription` property and methods serve the same purpose as their counterparts in the `TProjectDescriptor` class.

The `CreateSource` method is called by the IDE to create the source for the file: The result should be the contents of the (pascal unit) file. The parameters passed are the filename, source file name and resource name (if any) that the IDE has assigned to this file. A descendent should override this method to return the text of the source for this file.

The `GetResourceSource` method should be overridden by descendents to return the contents of the initial `.lfm` file (as text) that matches the source file. It will only be called if the `ResourceClass` class pointer is not `Nil`, i.e. when the IDE has decided that this is a form or datamodule or any visual object that needs streaming.

The meaning of the various properties should be straightforward:

DefaultFilename a default filename for this kind of file (with extension).

DefaultFileExt a default extension for this kind of file.

DefaultSourceName the default unit name.

DefaultResFileExt default extension for resource file (.lrs).

DefaultResourceName default name for the resource (`Form`).

RequiredPackages required packages for this kind of unit.

ResourceClass the class that the IDE will create when displaying the form for this file. Only when this property is not `Nil` will the IDE decide that a form (or datamodule) must be displayed when loading this file.

IsComponent Should be set to `True` if `ResourceClass` is a `TComponent` Descendent.

UseCreateFormStatements should be set to `True` if the IDE can add this resource to the `Application.CreateForm` statements.

IsPascalUnit should be set to `True` if this file is a pascal unit.

AddToProject should be set to true if a file of this kind should be added to the project when it is created.

For the project templates, the descendent is quite simple. It needs to override only the minimum of calls:

```
TProjectFileDesc=class(TProjectFileDescriptor)
  constructor Create(ATemplate : TProjectTemplate;
                     Values: TStrings);
  Function GetResourceSource : String;override;
```

```
  function CreateSource(const Filename, SourceName,
                        ResourceName: string): string; override;
  function GetLocalizedName: string; override;
  function GetLocalizedDescription: string; override;
end;
```

The `GetLocalizedName` and `GetLocalizedDescription` return simply a string, and so will not be shown here. The constructor only saves the variables which are passed to it, and initializes the name:

```
constructor TProjectFileDesc.Create(ATemplate: TProjectTemplate;
                                    Values : TStrings);
begin
  Inherited Create;
  FTemplate:=ATemplate;
  FVariables:=Values;
  Name:='Regular File';
end;
```

The routines which do the actual work are the `CreateSource` routine:

```
function TProjectFileDesc.CreateSource(const Filename, SourceName,
  ResourceName: string): string;

Var
  L : Tstrings;

begin
  L:=TstringList.Create;
  try
    FTemplate.CreateFile(FIndex,L,FVariables);
    Result:=L.Text;
  Finally
    L.Free;
  end;
end;
```

It simply passes the work on to the `TProjectTemplate` class. The `GetResourceSource` does something similar:

```
function TProjectFileDesc.GetResourceSource: String;

Var
  L : Tstrings;
  FN : String;

begin
  Result:='';
  If (ResourceClass<>Nil) then
    begin
    L:=TstringList.Create;
    try
      FN:=ChangeFileExt(FTemplate.FileNames[FIndex],'.lfm');
      FTemplate.CreateFile(FN,L,FVariables);
```

```
      Result:=L.Text;
    Finally
      L.Free;
    end;
    end;
end;
```

Note that it checks whether `ResourceClass` is set, which is only the case if a form file exists. The `CreateStartFiles` method discussed above, has taken care of that.

7 Conclusion

In the above, we have shown how to create a custom project in the Lazarus IDE. In doing so, various interfaces of the IDE have been used: the project interface and the file interface. However, the scope has been rather limited:

- Only a ready-made TForm descendent was created, and no other resources (e.g. a TDatamodule) were created: the interface offers more possibilities than were used here.
- What is more, the lazarus IDE interface offers other ways of obtaining the same functionality for a package that implements project templates. This interface was only touched upon, an needs to be explored more deeply.
- The possibility of creating a whole new category of new items was left untouched; This could be used to display the various templates immediatly in the 'New' dialog.

Exploring these other functionalities will be left to a future contribution, in which the project template functionality will be rewritten. At the same time it will be shown that the interface offers the possibility of creating custom resources (forms/data modules).