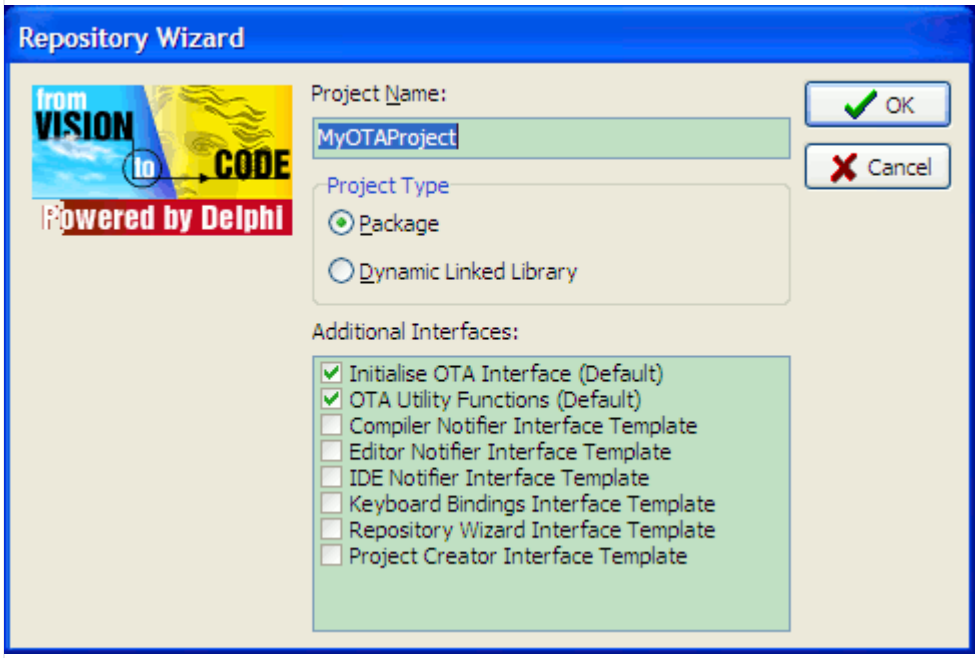
Dave's Development Blog
Software Development using Borland / Codegear / Embarcadero RAD Studio

Chapter 13: Project creators

By David | November 10, 2011 0 Comment

In this chapter I'm going to show you how to create projects in the IDE. In the next chapter I'll extend this to add modules to those projects.

Firstly, for the purposes of doing something useful we need a form which allow the user to select what they want to be created in a new project. We going to extend the project repository wizard so that you can create templates for Open Tools API projects. Below is the form that I've design and below that I'll explain some of the code behind it as this will help aid us in implementing the project and module creators.



In order to help with the configuration of this I've created some enumerates and sets as below. These define the types of project that can be generated (Package or DLL) and the modules that should be included in the project (based on the stuff we've covered so far).

```
TProjectType = (ptPackage, ptDLL);
TAdditionalModule = (
  amInitialiseOTAInterface,
```

```
    amUtilityFunctions,
    amCompilerNotifierInterface,
    amEditorNotifierInterface,
    amIDENotfierInterface,
    amKeybaordBindingsInterface,
    amReportioryWizardInterface,
    amProjectCreatorInterface
  );
  TAdditionalModules = Set Of TAdditionalModule;
```

Next we define a class method that can be used to invoke the dialogue, configure the form and finally return the users selected results as follows:

```
Class Function TfrmRepositoryWizard.Execute(var strProjectName : String;
  var enumProjectType : TProjectType;
  var enumAdditionalModules : TAdditionalModules): Boolean;

Const
  AdditionalModules : Array[Low(TAdditionalModule)..High(TAdditionalModule)] Of String = (
    'Initialise OTA Interface (Default)',
    'OTA Utility Functions (Default)',
    'Compiler Notifier Interface Template',
    'Editor Notifier Interface Template',
    'IDE Notifier Interface Template',
    'Keyboard Bindings Interface Template',
    'Repository Wizard Interface Template',
    'Project Creator Interface Template'
  );

Var
  i : TAdditionalModule;
  iIndex: Integer;

Begin
  Result := False;
  With TfrmRepositoryWizard.Create(Nil) Do
    Try
      edtProjectName.Text := 'MyOTAProject';
      rgpProjectType.ItemIndex := 0;
      // Default Modules
      enumAdditionalModules := [amInitialiseOTAInterface..amUtilityFunctions];
      For i := Low(TAdditionalModule) To High(TAdditionalModule) Do
        Begin
          iIndex := lbxAdditionalModules.Items.Add(AdditionalModules[i]);
          lbxAdditionalModules.Checked[iIndex] := i In enumAdditionalModules;
        End;
      If ShowModal = mrOK Then
        Begin
          strProjectName := edtProjectName.Text;
          Case rgpProjectType.ItemIndex Of
            0: enumProjectType := ptPackage;
```

```
            1: enumProjectType := ptDLL;
          End;
          For i := Low(TAdditionalModule) To High(TAdditionalModule) Do
            If lbxAdditionalModules.Checked[Integer(i)] Then
              Include(enumAdditionalModules, i)
            Else
              Exclude(enumAdditionalModules, i);
          Result := True;
        End;
    Finally
      Free;
    End;
End;
```

I find the use of enumerates and sets in the manner set out above a very useful and flexible way of configuring boolean options as it becomes very easy to add another option without having to reconfigure the dialogue with check boxes.

Next we need to validate the input of the form so that we do not get erroneous information. This is achieved in 3 parts. Firstly, an **OnClickCheck** event handler for the checked list box to ensure that the first 2 modules are always checked as they are needed for all other modules:

```
procedure TfrmRepositoryWizard.lbxAdditionalModulesClickCheck(Sender: TObject);
begin
  // Always ensure the default modules are Checked!
  lbxAdditionalModules.Checked[0] := True;
  lbxAdditionalModules.Checked[1] := True;
end;
```

Next there is an OnKeyPress event handler for the edit box to allow only valid identifier characters as follows:

```
procedure TfrmRepositoryWizard.edtProjectNameKeyPress(Sender: TObject; var Key: Char);
begin
  {$IFNDEF D2009}
  If Not (Key In ['a'..'z', 'A'..'Z', '0'..'9', '_']) Then
  {$ELSE}
  If Not CharInSet(Key, ['a'..'z', 'A'..'Z', '0'..'9', '_']) Then
  {$ENDIF}
    Key := #0;
end;
```

Finally an **OnClick** event handler for the OK button to ensure the follow:

- To ensure that the project name is not null;

- To ensure the project name starts with a letter or underscore (identifier requirement);
- To ensure the project name does not already exist in the IDE (requirement of the IDE from 2009 onwards).

```
procedure TfrmRepositoryWizard.btnOKClick(Sender: TObject);

Var
  boolProjectNameOK: Boolean;
  PG : IOTAProjectGroup;
  i: Integer;

begin
  If Length(edtProjectName.Text) = 0 Then
    Begin
      MessageDlg('You must specify a name for the project.', mtError, [mbOK], 0);
      ModalResult := mrNone;
      Exit;
    End;
  {$IFNDEF D2009}
  If edtProjectName.Text[1] In ['0'..'9'] Then
  {$ELSE}
  If CharInSet(edtProjectName.Text[1], ['0'..'9']) Then
  {$ENDIF}
    Begin
      MessageDlg('The project name must start with a letter or underscore.', mtError,
[mbOK], 0);
      ModalResult := mrNone;
      Exit;
    End;
  boolProjectNameOK := True;
  PG := ProjectGroup;
  For i := 0 To PG.ProjectCount - 1 Do
    If CompareText(ChangeFileExt(ExtractFileName(PG.Projects[i].FileName), ''),
      edtProjectName.Text) = 0 Then
      Begin
        boolProjectNameOK := False;
        Break;
      End;
  If Not boolProjectNameOK Then
    Begin
      MessageDlg(Format('There is already a project named "%s" in the project group!',
        [edtProjectName.Text]), mtError, [mbOK], 0);
      ModalResult := mrNone;
    End;
end;
```

The code for this can be found the the chapter download at the end of this article.

Next we need to look at the **IOTAProjectCreator** interface for creating the project itself. Below is the definition of a class that implements this interface (and its descendants):

```
  TProjectCreator = Class(TInterfacedObject, IOTACreator,IOTAProjectCreator
    {$IFDEF D0005}, IOTAProjectCreator50 {$ENDIF}
    {$IFDEF D0008}, IOTAProjectCreator80 {$ENDIF}
  )
  {$IFDEF D2005} Strict {$ENDIF} Private
    FProjectName : String;
    FProjectType : TProjectType;
  {$IFDEF D2005} Strict {$ENDIF} Protected
  Public
    Constructor Create(strProjectName : String; enumProjectType : TProjectType);
    // IOTACreator
    Function  GetCreatorType: String;
    Function  GetExisting: Boolean;
    Function  GetFileSystem: String;
    Function  GetOwner: IOTAModule;
    Function  GetUnnamed: Boolean;
    // IOTAProjectCreator
    Function  GetFileName: String;
    Function  GetOptionFileName: String; {$IFNDEF D0005} Deprecated; {$ENDIF}
    Function  GetShowSource: Boolean;
    Procedure NewDefaultModule; {$IFNDEF D0005} Deprecated; {$ENDIF}
    Function  NewOptionSource(Const ProjectName: String): IOTAFile; {$IFNDEF D0005}
Deprecated; {$ENDIF}
    Procedure NewProjectResource(Const Project: IOTAProject);
    Function  NewProjectSource(Const ProjectName: String): IOTAFile;
    {$IFDEF D0005}
    // IOTAProjectCreator50
    Procedure NewDefaultProjectModule(Const Project: IOTAProject);
    {$ENDIF}
    {$IFDEF D2005}
    // IOTAProjectCreator80
    Function  GetProjectPersonality: String;
    {$ENDIF}
  End;
```

This method is not part of any of the interfaces but is simply a constructor to save the project name and project type so that these can be passed to other functions during the creation process.

```
constructor TProjectCreator.Create(strProjectName: String; enumProjectType :
TProjectType);

begin
  FProjectName := strProjectName;
  FProjectType := enumProjectType;
end;
```

IOTACreator Methods

All the below methods are common to creators, i.e. will be required by Project and Module creators. These methods are called by the IDE as the item is being created.

The `GetCreatorType` method tells the IDE what type of information is to be returned. Since we are going to create the source ourselves then we return an empty string to signify this. If you want default source files generated by the IDE then you need to return the following strings for the following project types and return `NIL` from `NewProjectSource`.

- Package: `sPackage`;
- DLL: `sLibary`;
- GUI Project: `sApplication`;
- Console Project: `sConsole`.

```
function TProjectCreator.GetCreatorType: String;
begin
  Result := '';
end;
```

The `GetExisting` method tells the IDE is this is an existing project or a new project. We need a new project so we return `False`.

```
function TProjectCreator.GetExisting: Boolean;
begin
  Result := False;
end;
```

The `GetFileSystem` method returns the file system to be used. In our case were return an empty string for the default file system.

```
function TProjectCreator.GetFileSystem: String;
begin
  Result := '';
end;
```

The `GetOwner` method needs to return the project owner. In our case the current project group, so we pass it the result of our utility function **ProjectGroup**.

```
function TProjectCreator.GetOwner: IOTAModule;
```

```
begin
  Result := ProjectGroup;
end;
```

The `GetUnnamed` method determines whether the IDE will display the SaveAs dialogue on the first occasion when the file needs to be saved thus allowing the user to change the file name and path.

```
function TProjectCreator.GetUnnamed: Boolean;
begin
  Result := True;
end;
```

IOTAProjectCreator Methods

The below methods are common to Project Creators and are specific to the creation of a new project in the IDE. These methods are called by the IDE as the project is being created.

The `GetFileName` method must returns a fully qualified path for the module's file name. I made a mistake when coding this and did not append the file name with the correct file extension for the DLL and BPL files (i.e. .dpr and .dpk respectively). This caused the IDE to throw an access violation.

```
function TProjectCreator.GetFileName: String;
begin
  Case FProjectType Of
    ptPackage: Result := GetCurrentDir + '\' + FProjectName + '.dpk';
    ptDLL:     Result := GetCurrentDir + '\' + FProjectName + '.dpr';
  Else
    Raise Exception.Create('Unhandled project type in TProjectCreator.GetFileName.');
  End;
end;
```

The `GetOptionFileName` method is depreciated in later version of Delphi as the option information is stored in the DPROJ file rather than in separate DOF files. This method is to be used to specifying the DOF file.

```
function TProjectCreator.GetOptionFileName: String;
begin
  Result := '';
end;
```

The `GetShowSource` method simply tells the IDE whether to show the module source once created in the IDE.

```
function TProjectCreator.GetShowSource: Boolean;
begin
  Result := False;
end;
```

The `NewDefaultModule` method is a location where we can create the new modules for the project. Since is doesn't provide the project reference (`IOTAProject`) for the new project I will implement this elsewhere in the next chapter.

```
procedure TProjectCreator.NewDefaultModule;
begin
  //
end;
```

The `GetOptionSource` method allows you to specify the information in the options file defined above by returning a `IOTAFile` interface. For an example of how to do this please see below the method `NewProjectSource`.

```
function TProjectCreator.NewOptionSource(const ProjectName: String): IOTAFile;
begin
  Result := Nil;
end;
```

The `NewProjectResource` method allows you to create or modify the project resource associated with the passed `Project` reference.

```
procedure TProjectCreator.NewProjectResource(const Project: IOTAProject);
begin
  //
end;
```

Finally, the `NewProjectSource` method is where you can specify the custom source code for your project by returning a `IOTAFile` interface. We will cover this in a few minutes below.

```
function TProjectCreator.NewProjectSource(const ProjectName: String): IOTAFile;
begin
  Result := TProjectCreatorFile.Create(FProjectName, FProjectType);
```

```
end;
```

IOTAProjectCreator50 Methods

The below methods were introduced in Delphi 5. This method is meant to supersede the **NewDefaultModule**. This is where in the next chapter we will create the modules required for this OTA project.

```
{$IFDEF D0005}
procedure TProjectCreator.NewDefaultProjectModule(const Project: IOTAProject);
begin
  //
end;
{$ENDIF}
```

IOTAProjectCreator80 Methods

The below methods were introduced in Delphi 2005. This method is required to define the IDE personality under which the project is created as in the Galileo IDEs from 2005, multiple languages are supported. The function should return one of the pre-defined **sXxxxxxPersonality** strings defiend in ToolsAPI.pas as below:

- Delphi: **sDelphiPersonality**;
- Delphi .NET: **sDelphiDotNetPersonality**;
- C++ Builder: **sCBuilderPersonality**;
- C#: **sCSharpPersonality**;
- Visual Basic: **sVBPersonality**;
- Design: **sDesignPersonality**;
- Generic: **sGenericPersonality**.

Please note that not all of these personalities are available in later version of the IDE.

```
{$IFDEF D2005}
function TProjectCreator.GetProjectPersonality: String;
begin
  Result := sDelphiPersonality;
end;
{$ENDIF}
```

Above we said in the **NewProjectSource** method that we needed to return a **IOTAFile** interface for the new custom source code. In order to do this we need to create an instance of a class which implements the **IOTAFile** interface as follows:

```
  TProjectCreatorFile = Class(TInterfacedObject, IOTAFile)
  {$IFDEF D2005} Strict {$ENDIF} Private
    FProjectName : String;
    FProjectType : TProjectType;
  Public
    Constructor Create(strProjectName : String; enumProjectType : TProjectType);
    function GetAge: TDateTime;
    function GetSource: string;
  End;
```

The `IOTAFile` interface as 2 methods as below that need to be implemented and which are called by the IDE during creation:

The `Create` method here is simply a constructor that allows us to store information in the class for generating the source code.

```
constructor TProjectCreatorFile.Create(strProjectName: String; enumProjectType :
TProjectType);
begin
  FProjectName := strProjectName;
  FProjectType := enumProjectType;
end;
```

The `GetAge` is to return the file age of the source code. For our purposes we will return -1 signifying that the file has not been saved and is a new file.

```
function TProjectCreatorFile.GetAge: TDateTime;
begin
  Result := -1;
end;
```

The `GetSource` method does the heart of the work for the creation of a new project source. Here I've stored a text file of the project source for both libraries and packages in the plugins resource file (see previous posts on how this is achieved or the code at the end of the article). We extract the source from the resource file (with a unique name) and put it in a stream. We then convert the stream to a string. Note this is done in 2 different ways here due to me catering for non-Unicode and Unicode versions of Delphi.

```
function TProjectCreatorFile.GetSource: string;

Const
```

```
  strProjectTemplate : Array[Low(TProjectType)..High(TProjectType)] Of String = (
    'OTAProjectPackageSource', 'OTAProjectDLLSource');

ResourceString
  strResourceMsg = 'The OTA Project Template ''%s'' was not found.';

Var
  Res: TResourceStream;
  {$IFDEF D2009}
  strTemp: AnsiString;
  {$ENDIF}

begin
  Res := TResourceStream.Create(HInstance, strProjectTemplate[FProjectType], RT_RCDATA);
  Try
    If Res.Size = 0 Then
      Raise Exception.CreateFmt(strResourceMsg, [strProjectTemplate[FProjectType]]);
    {$IFNDEF D2009}
    SetLength(Result, Res.Size);
    Res.ReadBuffer(Result[1], Res.Size);
    {$ELSE}
    SetLength(strTemp, Res.Size);
    Res.ReadBuffer(strTemp[1], Res.Size);
    Result := String(strTemp);
    {$ENDIF}
  Finally
    Res.Free;
  End;
  Result := Format(Result, [FProjectName]);
end;
```

Now we have the code to implement the new project sources we need to tell the IDE how to invokes this. The below code is a modified version of the **Execute** method from the Repository Wizard Interface which displays the custom form we've created for asking the user what they want and then calls a new method **CreateProject** with the returned values.

```
Procedure TRepositoryWizardInterface.Execute;

Var
  strProjectName : String;
  enumProjectType : TProjectType;
  enumAdditionalModules : TAdditionalModules;

Begin
  If TfrmRepositoryWizard.Execute(strProjectName, enumProjectType,
    enumAdditionalModules) Then
    CreateProject(strProjectname, enumProjectType, enumAdditionalModules);
End;
```

Finally, the implementation of `CreateProject` below creates the project in the IDE.

```
procedure TRepositoryWizardInterface.CreateProject(strProjectName : String;
  enumProjectType : TProjectType; enumAdditionalModules : TAdditionalModules);

Var
  P: TProjectCreator;

begin
  P := TProjectCreator.Create(strProjectName, enumProjectType);
  FProject := (BorlandIDEServices As IOTAModuleServices).CreateModule(P) As IOTAProject;
end;
```

Now we can create either Packages or DLLs for our Open Tools API plugins.

All the files associated with this and all previous chapters can be downloaded here (OTAChapter13.zip).

Hope this is useful.

Dave 🙂

Category: Open Tools API Tags: Borland, BorlandIDEServices, CodeGear, Delphi, Embarcadero, Experts, IOTACreator, IOTAFile, IOTAModuleServices, IOTAProjectCreator, IOTAProjectCreator50, IOTAProjectCreator80, IOTAProjectGroup, IOTAWizard, OTA, RAD Studio