

Dave's Development Blog

Software Development using Borland / Codegear / Embarcadero RAD Studio

Chapter 14: Unit Creators

By David | November 16, 2011

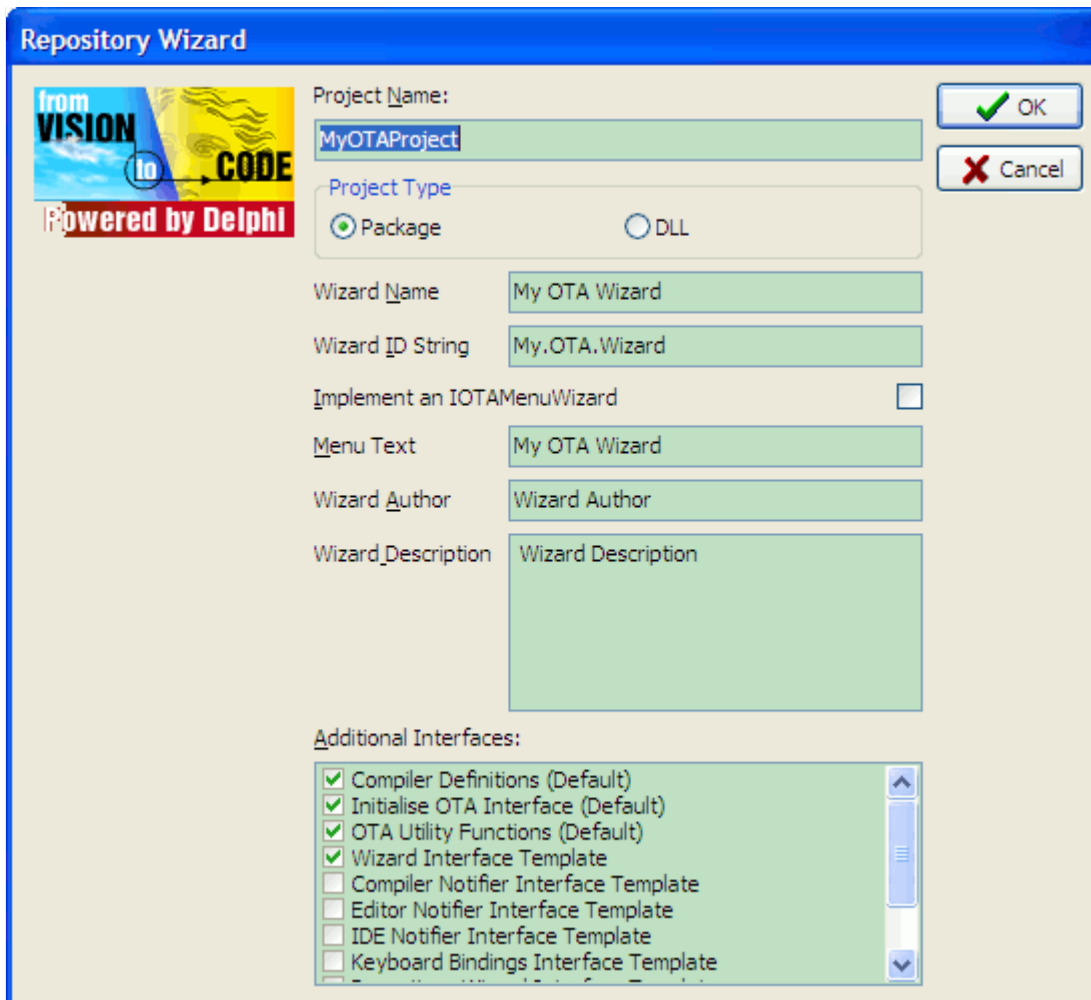
2 Comments

In this chapter I will finish what I started last time and provide the Module Creator code to create new projects in the IDE or more specifically new Open Tools API projects in the IDE.

While I was testing this code I noticed that the IDE did not maintain any custom code with a DPK packages. It seems the IDE maintains all the code in this file so it is futile to try and put custom code in there. I've seen this before when trying to conditionally compile using in a Package definition.

Repository Wizard Form

Below is a revised form. When I came to start implementing the full wizard I realised that there should be more information gathered at this stage so the amendments to the form are shown below and I'll explain the changes in the code below the image.



I've changed the definition of the enumerates to add in things that were missing and provided a new record which is used to pass data around instead of creating functions with long parameter list as shown below:

```

type
type
  TProjectType = (
    //ptApplication,
    ptPackage,
    ptDLL
  );
  TAdditionalModule = (
    amCompilerDefintions,
    amInitialiseOTAInterface,
    amUtilityFunctions,
    amWizardInterface,
    amCompilerNotifierInterface,
    amEditorNotifierInterface,
    amIDENotifierInterface,
    amKeyboardBindingInterface,
    amRepositoryWizardInterface,
    amProjectCreatorInterface,
    amModuleCreatorInterface
  );

```

```

TAdditionalModules = Set Of TAdditionalModule;

TProjectWizardInfo = Record
    FProjectName      : String;
    FProjectType      : TProjectType;
    FAdditionalModules : TAdditionalModules;
    FWizardName       : String;
    FWizardIDString   : String;
    FWizardMenu       : Boolean;
    FWizardMenuText   : String;
    FWizardAuthor     : String;
    FWizardDescription : String;
End;

```

Consequently the signature of the **Execute** method has changed below while the body of the code contains a few more assignments from edit boxes to fields of the record.

```

Class Function TfrmRepositoryWizard.Execute(var ProjectWizardInfo : TProjectWizardInfo):
Boolean;

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

Var
    i : TAdditionalModule;
    iIndex: Integer;
    j: TProjectType;

Begin
    Result := False;
    With TfrmRepositoryWizard.Create(Nil) Do
        Try
            rgpProjectType.Items.Clear;
            For j := Low(TProjectType) To High(TProjectType) Do

```

```

    rgpProjectType.Items.Add(ProjectTypes[j]);
edtProjectName.Text := 'MyOTAPProject';
rgpProjectType.ItemIndex := 0;
edtWizardName.Text := 'My OTA Wizard';
edtWizardIDString.Text := 'My.OTA.Wizard';
edtWizardMenuText.Text := 'My OTA Wizard';
edtWizardAuthor.Text := 'Wizard Author';
memWizardDescription.Text := 'Wizard Description';
// Default Modules
ProjectWizardInfo.FAdditionalModules := [amCompilerDefintions..amWizardInterface];
For i := Low(TAdditionalModule) To High(TAdditionalModule) Do
    Begin
        iIndex := lbxAdditionalModules.Items.Add(AdditionalModules[i]);
        lbxAdditionalModules.Checked[iIndex] := i In
ProjectWizardInfo.FAdditionalModules;
    End;
If ShowModal = mrOK Then
    Begin
        ProjectWizardInfo.FProjectName := edtProjectName.Text;
        ProjectWizardInfo.FProjectType := TProjectType(rgpProjectType.ItemIndex);
        ProjectWizardInfo.FWizardName := edtWizardName.Text;
        ProjectWizardInfo.FWizardIDString := edtWizardIDString.Text;
        ProjectWizardInfo.FWizardMenu := cbxMenuWizard.Checked;
        ProjectWizardInfo.FWizardMenuText := edtWizardMenuText.Text;
        ProjectWizardInfo.FWizardAuthor := edtWizardAuthor.Text;
        ProjectWizardInfo.FWizardDescription := memWizardDescription.Text;
        For i := Low(TAdditionalModule) To High(TAdditionalModule) Do
            If lbxAdditionalModules.Checked[Integer(i)] Then
                Include(ProjectWizardInfo.FAdditionalModules, i)
            Else
                Exclude(ProjectWizardInfo.FAdditionalModules, i);
            Result := True;
        End;
    Finally
        Free;
    End;
End;

```

The [ClickCheck](#) method has been updated as there are 4 default modules for an OTA project.

```

procedure TfrmRepositoryWizard.lbxAdditionalModulesClickCheck(Sender: TObject);

Var
    iModule: TAdditionalModule;

begin
    For iModule := amCompilerDefintions To amWizardInterface Do
        lbxAdditionalModules.Checked[Integer(iModule)] := True;
    end;

```

Finally the OK button's **OnClick** event handler has been modified to check that the input of the additional dialogue controls is valid.

```

procedure TfrmRepositoryWizard.btnOKClick(Sender: TObject);

Procedure CheckTextField(strText, strMsg : String);

Begin
    If strText = '' Then
        Begin
            MessageDlg(strMsg, mtError, [mbOK], 0);
            ModalResult := mrNone;
            Abort;
        End;
    End;

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;
    {$IFDEF D2009}
    If edtProjectName.Text[1] In ['0'..'9'] Then
        {$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;
CheckTextField(edtWizardName.Text, 'You must specify a Wizard Name.');
```

```

CheckTextField(edtWizardIDString.Text, 'You must specify a Wizard ID String.');
```

```

CheckTextField(edtWizardMenuText.Text, 'You must specify a Wizard Menu Text.');
```

```

end;
```

Module Creator

Now we moved on to the definition of the Module Creator. This is very similar to the Project Creator from the last chapter and the methods of the interfaces are called by the IDE when the module is created.

```

TModuleCreator = Class(TInterfacedObject, IOTACreator, IOTAModuleCreator)
{$IFDEF D2005} Strict {$ENDIF} Private
    FProject          : IOTAProject;
    FProjectWizardInfo : TProjectWizardInfo;
    FAdditionalModule  : TAdditionalModule;
{$IFDEF D2005} Strict {$ENDIF} Protected
Public
    Constructor Create(AProject : IOTAProject; ProjectWizardInfo : TProjectWizardInfo;
        AdditionalModule : TAdditionalModule);
    // IOTACreator
    Function GetCreatorType: String;
    Function GetExisting: Boolean;
    Function GetFileSystem: String;
    Function GetOwner: IOTAModule;
    Function GetUnnamed: Boolean;
    // IOTAModuleCreator
    Procedure FormCreated(Const FormEditor: IOTAFormEditor);
    Function GetAncestorName: String;
    Function GetFormName: String;
    Function GetImplFileName: String;
    Function GetIntfFileName: String;
    Function GetMainForm: Boolean;
    Function GetShowForm: Boolean;
    Function GetShowSource: Boolean;
    Function NewFormFile(Const FormIdent: String; Const AncestorIdent: String) : IOTAFile;
    Function NewImplSource(Const ModuleIdent: String; Const FormIdent: String;
        Const AncestorIdent: String): IOTAFile;
    Function NewIntfSource(Const ModuleIdent: String; Const FormIdent: String;
        Const AncestorIdent: String): IOTAFile;
End;
```

I'll talk about the methods further down the page.

Since we are going to specify our own code we will also need the file creator below:

```

TModuleCreatorFile = Class(TInterfacedObject, IOTAFile)
{$IFDEF D2005} Strict {$ENDIF} Private
    FProjectWizardInfo : TProjectWizardInfo;
    FAdditionalModule   : TAdditionalModule;
{$IFDEF D2005} Strict {$ENDIF} Protected
    Function ExpandMacro(strText, strMacroName, strReplaceText : String) : String;
    Function GetFinaliseWizardCode : String;
    Function GetInitialiseWizardCode : String;
    Function GetVariableDeclCode : String;
    Function GetUsesClauseCode : String;
Public
    Constructor Create(ProjectWizardInfo : TProjectWizardInfo;
        AdditionalModule : TAdditionalModule);
    function GetAge: TDateTime;
    function GetSource: string;
End;

```

I'll talk about the methods further down the page.

In order to make this easier I've used text files in the projects resources to hold the source code which in turn contains some "macros" that get expanded during creation, (for example `$MODULENAME$` is for the module name of the unit).

In order to do this I've defined a simple record which allows us to create a static constant array containing all the information we need for resource names and module names that can be called depending on the `TAdditionalModule` enumerate passed.

Type

```

TModuleInfo = Record
    FResourceName : String;
    FModuleName   : String;
End;

```

Const

```

strProjectTemplate : Array[Low(TAdditionalModule)..High(TAdditionalModule)] of
TModuleInfo = (
    (FResourceName: 'OTAModuleCompilerDefinitions';      FModuleName:
'CompilerDefinitions.inc'),
    (FResourceName: 'OTAModuleInitialiseOTAInterfaces';  FModuleName:
'InitialiseOTAInterface.pas'),
    (FResourceName: 'OTAModuleUtilityFunctions';        FModuleName:
'UtilityFunctions.pas'),
    (FResourceName: 'OTAModuleWizardInterface';         FModuleName:
'WizardInterface.pas'),
    (FResourceName: 'OTAModuleCompilerNotifierInterface'; FModuleName:
'CompilerNotifierInterface.pas'),
    (FResourceName: 'OTAModuleEditorNotifierInterface';  FModuleName:
'EditorNotifierInterface.pas'),

```

```

        (FResourceName: 'OTAModuleIDENotifierInterface'; FModuleName:
'IDENotifierInterface.pas'),
        (FResourceName: 'OTAModuleKeyboardBindingInterface'; FModuleName:
'KeyboardBindingInterface.pas'),
        (FResourceName: 'OTAModuleRepositoryWizardInterface'; FModuleName:
'RepositoryWizardInterface.pas'),
        (FResourceName: 'OTAModuleProjectCreatorInterface'; FModuleName:
'ProjectCreatorInterface.pas'),
        (FResourceName: 'OTAModuleModuleCreatorInterface'; FModuleName:
'ModuleCreatorInterface.pas')
    );

```

IOTACreator

The methods of the [IOTACreator](#) interface are the same as those defined in the Project Creator.

Firstly, we define a constructor so that we can pass the project and project information on to the file creator class when actually creating the modules.

```

constructor TModuleCreator.Create(AProject: IOTAProject; ProjectWizardInfo :
TProjectWizardInfo;
    AdditionalModule : TAdditionalModule);
begin
    FProject := AProject;
    FProjectWizardInfo := ProjectWizardInfo;
    FAdditionalModule := AdditionalModule;
end;

```

The [GetCreatorType](#) should return the string representing the type of module to create. This can be any one of the following strings:

- [sUnit](#): Unit module;
- [sForm](#): Form Module;
- [sText](#): RAW text module with no code.

In this instance we return [sUnit](#) as we need units for our interface modules.

```

function TModuleCreator.GetCreatorType: String;
begin
    Result := sUnit;
end;

```

The [GetExisting](#) method tells the IDE if this is an existing module. We are creating a new one so will return

false.

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

The `GetFileSystem` method should return an empty string inferring that we are using the default file system.

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

The `GetOwner` method should return the project to which the module should be associated. In this case we return the project passed in the class's constructor.

```
function TModuleCreator.GetOwner: IOTAModule;  
begin  
    Result := FProject;  
end;
```

Finally, the `GetUnnamed` method should return True to signify that this is a new unsaved module and therefore the IDE should ask the user on the first time of saving as to where they would like to save the file and possibly rename it.

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

IOTAModuleCreator

Now for the methods of the module creator which again are called by the IDE on creation of the module. Note: this interface has not changed and therefore there are no numbered interfaces to implement for different version of Delphi.

The `FormCreated` method is called once the new form or data module has been created so that you can manipulate the form by adding controls.

```
procedure TModuleCreator.FormCreated(const FormEditor: IOTAFormEditor);
begin
end;
```

The `GetAncestorName` method as far as I'm aware is only called if you are creating a form and this is used as the ancestor for the form.

```
function TModuleCreator.GetAncestorName: String;
begin
    Result := 'TForm';
end;
```

The `GetFormName` should return the name of the form when you are creating a form. In the case of a unit this is ignored.

```
function TModuleCreator.GetFormName: String;
begin
    { Return the form name }
    Result := 'MyForm1';
end;
```

The `GetImplFileName` method should return the name of the implementation file (.pas file in Delphi or .cpp file in C++, etc). This must be a fully qualified `drive:\path\filename.ext`. You can leave this blank to have the IDE create a new unique one for you.

```
function TModuleCreator.GetImplFileName: String;
begin
    Result := GetCurrentDir + '\' + strProjectTemplate[FAdditionalModule].FModuleName;
end;
```

The `GetIntfFileName` method is only applicable to C++ as Delphi .pas files have their interface section within them. Therefore return an empty string for the IDE to handle this itself.

```
function TModuleCreator.GetIntfFileName: String;
```

```
begin
  Result := '';
end;
```

The `GetMainForm` method should return true when creating a form IF this will be the projects main form. For our exercise this can be false.

```
function TModuleCreator.GetMainForm: Boolean;
begin
  Result := False;
end;
```

The `GetShowForm` method should return true if you want the form to be displayed once created. For our purposes this can be false.

```
function TModuleCreator.GetShowForm: Boolean;
begin
  Result := False;
end;
```

The `GetShowSource` method should return true if you want the unit to be displayed once created. For our purposes this can be true.

```
function TModuleCreator.GetShowSource: Boolean;
begin
  Result := True;
end;
```

The `NewFormFile` method is where you can provide the source to the DFM file and create you own form. For our purposes this will return `Nil`.

```
function TModuleCreator.NewFormFile(const FormIdent, AncestorIdent: String): IOTAFile;
begin
  Result := Nil;
end;
```

The `NewImplSource` method is where we return a `IOTAFile` interface to create our custom source code for our modules in the same manner as we did for the Project Creator.

```
function TModuleCreator.NewImplSource(const ModuleIdent, FormIdent,
  AncestorIdent: String): IOTAFile;
begin
  Result := TModuleCreatorFile.Create(FProjectWizardInfo, FAdditionalModule);
end;
```

The `NewIntfSource` method is where we would return a `IOTAFile` interface to create a C++ interface header file. For our example we don't need this so will return `Nil`.

```
function TModuleCreator.NewIntfSource(const ModuleIdent, FormIdent,
  AncestorIdent: String): IOTAFile;
begin
  Result := Nil;
end;
```

TModuleCreatorFile

The implementation of the file creator is essentially the same as that of the project creator however I've implemented the ability to expand "macros" in the templates to replace for example `$MODULENAME$` with the name of the module.

The below method is a simple constructor to allow use to pass the project wizard information and the specific type of module being created to the `GetSource` method.

```
constructor TModuleCreatorFile.Create(ProjectWizardInfo : TProjectWizardInfo;
  AdditionalModule : TAdditionalModule);
begin
  FProjectWizardInfo := ProjectWizardInfo;
  FAdditionalModule := AdditionalModule;
end;
```

The `GetAge` method returns -1 to signify that this is a new unsaved file.

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

The `GetSource` method is where we return the source code for each module through the use of the constant

array defined earlier and the **AdditionalModule** parameter passed to the class's constructor.

```
function TModuleCreatorFile.GetSource: string;

Const
  WizardMenu : Array[False..True] Of String = ('', ' ', IOTAMenuWizard');

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

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

begin
  Res := TResourceStream.Create(HInstance,
  strProjectTemplate[FAdditionalModule].FResourceName,
  RT_RCDATA);
  Try
    If Res.Size = 0 Then
      Raise Exception.CreateFmt(strResourceMsg,
      [strProjectTemplate[FAdditionalModule].FResourceName]);
    {$IFDEF D2009}
    SetLength(Result, Res.Size);
    Res.ReadBuffer(Result[1], Res.Size);
    {$ELSE}
    SetLength(strTemp, Res.Size);
    Res.ReadBuffer(strTemp[1], Res.Size);
    Result := String(strTemp);
    {$ENDIF}
  Finally
    Res.Free;
  End;
  Result := ExpandMacro(Result, '$MODULENAME$',
  ChangeFileExt(strProjectTemplate[FAdditionalModule].FModuleName, ''));
  Result := ExpandMacro(Result, '$USESCLAUSE$', GetUsesClauseCode);
  Result := ExpandMacro(Result, '$VARIABLEDECL$', GetVariableDeclCode);
  Result := ExpandMacro(Result, '$INITIALISEWIZARD$', GetInitialiseWizardCode);
  Result := ExpandMacro(Result, '$FINALISEWIZARD$', GetFinaliseWizardCode);
  Result := ExpandMacro(Result, '$WIZARDNAME$', FProjectWizardInfo.FWizardName);
  Result := ExpandMacro(Result, '$WIZARDIDSTRING$', FProjectWizardInfo.FWizardIDString);
  Result := ExpandMacro(Result, '$WIZARDMENUTEXT$', FProjectWizardInfo.FWizardMenuText);
  Result := ExpandMacro(Result, '$AUTHOR$', FProjectWizardInfo.FWizardAuthor);
  Result := ExpandMacro(Result, '$WIZARDDESCRIPTION$',
  FProjectWizardInfo.FWizardDescription);
  Result := ExpandMacro(Result, '$WIZARDMENUREQUIRED$',
  WizardMenu[FProjectWizardInfo.FWizardMenu]);
end;
```

This next function simply allow you to substitute a macro name in the given text with some other text and

have this returned by the function.

```
function TModuleCreatorFile.ExpandMacro(strText, strMacroName, strReplaceText: String):
String;

Var
    iPos : Integer;

begin
    iPos := Pos(LowerCase(strMacroName), LowerCase(strText));
    Result := strText;
    While iPos > 0 Do
        Begin
            Result :=
                Copy(strText, 1, iPos - 1) +
                strReplaceText +
                Copy(strText, iPos + Length(strMacroName), Length(strText) - iPos + 1 -
Length(strMacroName));
            iPos := Pos(LowerCase(strMacroName), LowerCase(Result));
        End;
    end;
```

The below function returns a string of code that needs to be inserted into the Finalization section of the main unit to remove the selected wizards from the IDE.

```
function TModuleCreatorFile.GetFinaliseWizardCode: String;
begin
    If amKeyboardBindingInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
            ' // Remove Keyboard Binding Interface'#13#10 +
            ' If iKeyBindingIndex > iWizardFailState Then'#13#10 +
            '     (BorlandIDEServices As
IOTAKeyboardServices).RemoveKeyboardBinding(iKeyBindingIndex);'#13#10;
    If amIDENotifierInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
            ' // Remove IDE Notifier Interface'#13#10 +
            ' If iIDENotfierIndex > iWizardFailState Then'#13#10 +
            '     (BorlandIDEServices As IOTAServices).RemoveNotifier(iIDENotfierIndex);'#13#10;
    If amCompilerNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
            ' {$IFDEF D2010}'#13#10 +
            ' // Remove Compiler Notifier Interface'#13#10 +
            ' If iCompilerIndex <> iWizardFailState Then'#13#10 +
            '     (BorlandIDEServices As
IOTACompileServices).RemoveNotifier(iCompilerIndex);'#13#10 +
            ' {$ENDIF}'#13#10;
    If amEditorNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
            ' {$IFDEF D2005}'#13#10 +
```

```

    ' // Remove Editor Notifier Interface'#13#10 +
    ' If iEditorIndex <> iWizardFailState Then'#13#10 +
    '     (BorlandIDEServices As
IOTAEditorServices).RemoveNotifier(iEditorIndex);'#13#10 +
    ' {$ENDIF}'#13#10;
If amRepositoryWizardInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result +
    ' // Remove Repository Wizard Interface'#13#10 +
    ' If iRepositoryWizardIndex <> iWizardFailState Then'#13#10 +
    '     (BorlandIDEServices As
IOTAWizardServices).RemoveWizard(iRepositoryWizardIndex);'#13#10;
end;

```

The below function returns a string of code that needs to be inserted into the Initialization section of the main unit to create the selected wizards in the IDE.

```

function TModuleCreatorFile.GetInitialiseWizardCode: String;
begin
    If amKeyboardBindingInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
        ' // Create Keyboard Binding Interface'#13#10 +
        ' iKeyBindingIndex := (BorlandIDEServices As
IOTAKeyboardServices).AddKeyboardBinding('#13#10 +
        '     TKeybindingTemplate.Create);'#13#10;
    If amIDENotifierInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
        ' // Create IDE Notifier Interface'#13#10 +
        ' iIDENotifierIndex := (BorlandIDEServices As IOTAServices).AddNotifier('#13#10 +
        '     TIDENotifierTemplate.Create);'#13#10;
    If amCompilerNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
        ' {$IFDEF D2010}'#13#10 +
        ' // Create Compiler Notifier Interface'#13#10 +
        ' iCompilerIndex := (BorlandIDEServices As
IOTACompileServices).AddNotifier('#13#10 +
        '     TCompilerNotifier.Create);'#13#10 +
        ' {$ENDIF}'#13#10;
    If amEditorNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
        ' {$IFDEF D2005}'#13#10 +
        ' // Create Editor Notifier Interface'#13#10 +
        ' iEditorIndex := (BorlandIDEServices As IOTAEditorServices).AddNotifier('#13#10 +
        '     TEditorNotifier.Create);'#13#10 +
        ' {$ENDIF}'#13#10;
    If amRepositoryWizardInterface In FProjectWizardInfo.FAdditionalModules Then
        Result := Result +
        ' // Create Project Repository Interface'#13#10 +
        ' iRepositoryWizardIndex := (BorlandIDEServices As
IOTAWizardServices).AddWizard('#13#10 +
        '     TRepositoryWizardInterface.Create);'#13#10;
end;

```

The below function returns a string of code that needs to be inserted into the Uses clause of the main unit to allow access to the wizard interface definitions.

```
function TModuleCreatorFile.GetUsesClauseCode: String;
begin
  If amKeyboardBindingInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result + ' KeyboardBindingInterface, '#13#10;
  If amIDENotifierInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result + ' IDENotifierInterface, '#13#10;
  If amCompilerNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result + ' CompilerNotifierInterface, '#13#10;
  If amEditorNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result + ' EditorNotifierInterface, '#13#10;
  If amRepositoryWizardInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result + ' RepositoryWizardInterface, '#13#10;
end;
```

The below function returns a a string of code that needs to be inserted into the variable declaration section of the main unit to hold the indexes of the wizard created.

```
function TModuleCreatorFile.GetVariableDeclCode: String;
begin
  If amKeyboardBindingInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result +
      ' iKeyBindingIndex      : Integer = iWizardFailState; '#13#10;
  If amIDENotifierInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result +
      ' iIDENotfierIndex      : Integer = iWizardFailState; '#13#10;
  If amCompilerNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result +
      ' {$IFDEF D2010}'#13#10 +
      ' iCompilerIndex        : Integer = iWizardFailState; '#13#10 +
      ' {$ENDIF}'#13#10;
  If amEditorNotifierInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result +
      ' {$IFDEF D0006}'#13#10 +
      ' iEditorIndex           : Integer = iWizardFailState; '#13#10 +
      ' {$ENDIF}'#13#10;
  If amRepositoryWizardInterface In FProjectWizardInfo.FAdditionalModules Then
    Result := Result +
      ' iRepositoryWizardIndex : Integer = iWizardFailState; '#13#10;
end;
```

Updates to the Project Creator

Finally, we modify the New `DefaultProjectModule` code to iterate through all the different modules types creating those that have been selected.


```

{$IFDEF D0005}
procedure TProjectCreator.NewDefaultProjectModule(const Project: IOTAProject);

Var
  M: TModuleCreator;
  iModule: TAdditionalModule;

begin
  For iModule := Low(TAdditionalModule) To High(TAdditionalModule) Do
    If iModule In FProjectWizardInfo.FAdditionalModules Then
      Begin
        M := TModuleCreator.Create(Project, FProjectWizardInfo, iModule);
        (BorlandIDEServices As IOTAModuleServices).CreateModule(M);
      End;
    end;
  {$ENDIF}

```

I hope this is straight forward. The code for this chapter can be downloaded here ([OTChapter14.zip](#)). Enjoy



regards

Dave 😊

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

2 thoughts on “Chapter 14: Unit Creators”



Miguel

March 5, 2012


I found the Open Tools and find them very useful, but I found a problem. When I try to write NewFormFile event, not listen to me and change the text automatically. I have a class TBaseForm inherited from TForm, and THomeForm inherited from TBaseForm and when I create a THomeForm from IDE then I lose NewFormFile text ('inherited% s:% s end T') and I get "Object% s".

Best regards and many thanks!



David

[Post author](#)

 March 6, 2012

Miguel,

At the moment I can not actual create a working example but have you implemented GetAncestorName()?

The Delphi 7 help states the following: "Implement the GetAncestorName to return the name (not the type) of the ancestor form if you are using form inheritance. Return an empty string for an ordinary unit or form. Return "DataModule" to create a custom data module."

If this doesn't help can you provide the code that goes with your class which creates the form.

regards

Dave.

Comments are closed.

Iconic One Theme | Powered by Wordpress