Custom forms in Delphi

Michaël Van Canneyt

March 19, 2016

Abstract

This article describes how to put your own `TForm` or `TCustomForm` descendant in the "File - New menu" of the IDE. For this, the IDE's Open Tools API is used.

1 Introduction

When making a lot of applications or even just a few but large applications, there are inevitably a lot of tasks that will be re-implemented for each form:

- Warn about losing unsaved data when closing the form, optionally save pending changes.
- Export data.
- Refresh data.
- Print things.
- Save and restore the form's position.
- Remove properties that are not supposed to be used because they must be handled globally (e.g. scaled).

Doubtlessly many other common things can be found.

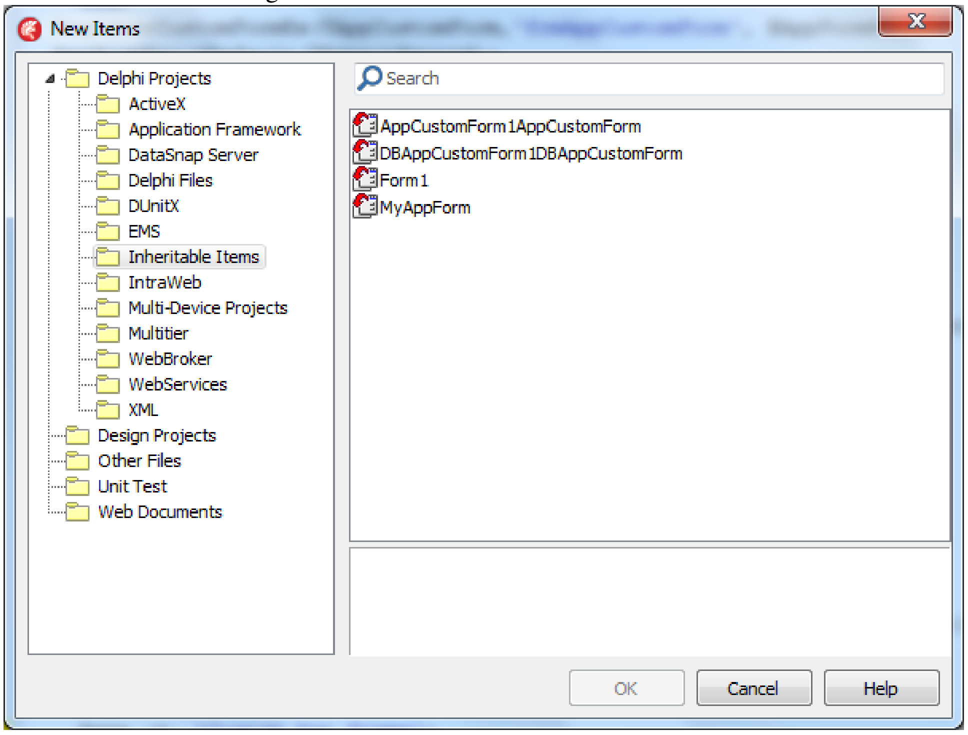
Since Delphi uses Object Pascal and a form is just a class as any other, it makes sense to implement this common functionality in a parent form: this is the basic idea of Visual Form inheritance: it is simple class inheritance. The Visual Form Inheritance dialog figure 1 on page 2 shows all the forms, frames or datamodules in a project and allows you to make a descendant from them in an easy and straightforward manner.

Visual Form inheritance has some drawbacks:

- The parent form needs to be part of the application, or must be put in a local repository directory. The former means it's difficult to share a form over projects, and the latter does not work very well when multiple developers are working remotely on the same project.
- It is not possible to implement custom properties and register property editors for the base form class.

This can be remedied by creating a `TCustomForm` or `TForm` descendant class in a package, and registering it in a design-time package in the IDE. A new instance of such a form can then be created from the 'File - new - other' menu. The advantage over Visual Form

Figure 1: Visual Form Inheritance in the IDE



Inheritance is that the package can be used in multiple projects, it can be distributed to all developers who can put it wherever they please.

The disadvantage (to the best of the knowledge of the author) is that the visual form inheritance can not be simulated using this technique: Visual Form Inheritance allows you to design a form, put components on them, customize these components. The descendants then will all have these components on them. For a registered descendant of `T(Custom)Form` which was designed visually with components on it, the same cannot be done: Although the components will be created at runtime, they will not be created when designing an instance of the form in the IDE designer. This means that all that can be done is create a custom form class in code, add properties and methods to it.

The mechanism described here can also be used for frames and datamodules.

2 The Open Tools API

Interaction with the IDE happens by means of the Open Tools API. This is a large API, and to implement custom forms, only a small part of the API is used. The API works through a number of interface declarations: the programmer implements some classes, each implementing one ore more desired interfaces, and registers the interface with the IDE.

To register custom forms in the IDE, a number of interfaces must be implemented. First of all, to display an item in the “File - New - Other” menu the following interfaces are needed:

IOTAWizard This is the basic interface for all wizards. Basically, it must return some strings to identify the wizard for the IDE, and has an `Execute` method.

IOTARepositoryWizard This interface is the basic interface needed to put an item in the

"File - New - Other" menu. It implements the necessary functionality so the IDE can show an item in the menu.

IOTARepositoryWizard60 This is an extension of the `IOTARepositoryWizard` interface to help the IDE to decide whether the item should be shown in VCL/FMX/CLX projects.

IOTARepositoryWizard80 This is an extension of the `IOTARepositoryWizard60` interface to help the IDE to decide in what personalities the item must be shown.

IOTARepositoryWizard160 This is an extension of the `IOTARepositoryWizard60` interface to help the IDE to decide for what framework/platform the items must be shown.

IOTAFormWizard This interface is empty, but its presence indicates to the IDE that the wizard creates a form.

For the latest versions of Delphi, it is best to implement `IOTARepositoryWizard160`.

The wizard must create a new (form) file in the current project. For this, again a number of interfaces must be implemented:

IOTACreator This is the basic interface for creating new items in a project.

IOTAModuleCreator This is an extension of `IOTACreator` to create modules (forms) in the IDE. Basically this creates a new entry in the project or project group file.

IOTAFile This interface is an extension of `IOTACreator` to create an actual file: a module can consist of multiple files: a form consists of 2 files: a form file and a pascal unit.

All these classes are present in the `ToolsAPI` unit.

Lastly, the IDE must be told that the new form class is a form that can be edited: the class must be registered in the IDE. This is done with the `RegisterCustomModule` call in the Open Tools API, in the unit `DesignIntf`.

3 A simpler API

To implement a class with all these interfaces every time one wishes to add a form to the IDE would not be very efficient. Therefor, a much simplified API is created and used. The xptFormWizards unit is used to implement this simple API.

This unit presents 2 calls to register a custom class form in the IDE:

```
Procedure RegisterCustomFormWizard(
    CompClass: TCustomComponentClass;
    Const Clause: String);
Procedure RegisterCustomFormWizardEx(
   CompClass: TCustomComponentClass;
   Const Clause: String;
   Options: PExpertRecord);
```

The `Clause` is a list of units that must be added to the uses clause of the unit that implements the custom class. The unit in which the class is implemented can be determined from the class pointer itself, but it can be interesting to extend this with other units.

The optional `Options` argument is a pointer to a record containing some extra information, which is self-explaining:

```
PExpertRecord = ^TExpertRecord;
TExpertRecord = Record
  Name    : String;
  Author  : String;
  Comment : String;
  Page    : String;
  Glyph   : Cardinal;
  IDString: String;
End;
```

The IDString must of course be unique when registering multiple forms. Defaults will be used for these options if they are not specified. These 2 calls can be easily used in a Design-time package to register a form class.

When a design-time package that registers a custom form is unloaded from the IDE, it must of course be remove the item again from the IDE's menu. The following 2 calls remove the class from the IDE:

```
Procedure UnRegisterCustomFormWizard(CompClass: TCustomComponentClass); overload;
Procedure UnRegisterCustomFormWizard(Const CompClassName: String); overload;
```

4 Implementing the repository Wizard

The `RegisterCustomFormWizardEx` class creates a class `TMKFormWizardExpert` which implements `IOTARepositoryWizard` and registers it in the IDE. It takes some precautions so it does not registers the same class twice: it keeps a stringlist with the classnames of all form, frame or datamodule classes it registers, and checks that a form is not registered twice:

```
Procedure RegisterCustomFormWizardEx(
  CompClass: TCustomComponentClass;
  Const Clause: String;
  Options: PExpertRecord);

Var
   AClassName: String;
   Tmp: TExpertRecord;
   Wizard : TMKFormWizardExpert;

Begin
  If (CompClass=TCustomForm) Or (CompClass=TForm) Or
     (CompClass=TCustomFrame) Or (CompClass=TFrame) Or
     (CompClass=TDatamodule) Then
    Exit;
  AClassName := CompClass.ClassName;
  // Check if it was registered before.
  if FormWizardClassList.IndexOf(AClassName)<>-1  then
    // it was registered, just update the class pointer.
    RegisterCustomModule(CompClass, TxptFormWizardModule)
  else
    begin
    // It was not yet registered, register it.
    RegisterCustomModule(CompClass, TxptFormWizardModule);
    If (Options=Nil) Then
       Begin
       Tmp:=Default(TExpertRecord);
       Tmp.Name:=AClassName;
       Options:=@Tmp;
       end;
    // Here the actual registration happens:
    Wizard := TMKFormWizardExpert.Create(CompClass, Clause, Options^);
    With BorlandIDEServices as IOTAWizardServices do
      Wizard.WizardIndex := AddWizard(Wizard);
    // And here we add it to the list
    FormWizardClassList.AddObject(AClassName, Wizard);
    end;
End;
```

Most of this code simply consists of some error checking and creating an options record if none was specified. The statement

```
  With BorlandIDEServices as IOTAWizardServices do
    Wizard.WizardIndex := AddWizard(Wizard);
```

actually registers the form in the IDE's "File" menu.

The `TMKFormWizardExpert` is the class that actually implements the `IOTARepositoryWizard` and some other interfaces:

```
TMKFormWizardExpert = Class(TInterfacedObject, IOTAWizard,
  IOTANotifier, IOTARepositoryWizard, IOTAFormWizard,
  IOTARepositoryWizard80)
Private
  FBaseClass    : TCustomComponentClass;
Public
  Constructor Create(Ancestor: TCustomComponentClass;
                     Const Clause: String;
                     Options: TExpertRecord); Virtual;
{ IOTAWizard }
  Function GetIDString: String;
  Function GetName: String;
  Function GetState: TWizardState;
  Procedure Execute;
  { IOTANotifier }
  Procedure AfterSave;
  Procedure BeforeSave;
  Procedure Destroyed;
  Procedure Modified;
  { IOTARepositoryWizard / IOTAFormWizard }
  Function GetAuthor: String;
  Function GetComment: String;
  Function GetPage: String;
  Function GetGlyph: cardinal;
  { IOTARepositoryWizard80 }
  function GetDesigner: string;
  function GetGalleryCategory: IOTAGalleryCategory;
  function GetPersonality: string;
end;
```

Many of the methods listed here just return the various fields in `TExpertRecord`, or some default value. The only interesting method is the `Execute` method.

The `Execute` method of this class is called when the user chooses the wizard in the "File - New - Other" dialog.

In the `Execute` method, we use the `IOTAModuleServices` interface from the IDE, more specifically the `CreateModule` method, to create a new form:

```
Procedure TMKFormWizardExpert.Execute;
var
  M : TMKFormWizardModuleCreator;

begin
  // the  TMKFormWizardModuleCreator does the actual work.
  M:=TMKFormWizardModuleCreator.Create(FBaseClass,
                                       FClause,
                                       FUseInherited);
  With (BorlandIDEServices As IOTAModuleServices) Do
    CreateModule(M As IOTACreator);
end;
```

What is shown here is of course quite simple and straightforward. One of the things that can be done here is to show a dialog to customize the generated sources: maybe ask for some default values for properties to be saved in the form file, override virtual methods that exist in the parent class, etc. All this should be done in the `Execute` method, and the response of the user can be passed on to the `TMKFormWizardModuleCreator` class which will do the actual work.

5 Creating the sources for the new form

The `TMKFormWizardModuleCreator` class introduced above is responsible for creating the initial sources for the new form; the IDE will call its various methods:

```
TMKFormWizardModuleCreator = Class(TInterfacedObject,
   IOTACreator,
   IOTAModuleCreator)
Public
  Constructor Create(Ancestor: TCustomComponentClass;
                     Const Clause: String;
                     useInherited : Boolean);  Virtual;
{ IOTACreator }
  Function GetCreatorType: String;
  Function GetExisting: Boolean;
  Function GetFileSystem: String;
  Function GetOwner: IOTAModule;
  Function GetUnNamed: Boolean;
{ IOTAModuleCreator }
  Function GetAncestorName: String;
  Function GetImplFileName: String;
  Function GetIntfFileName: String;
  Function GetFormName: String;
  Function GetMainForm: Boolean;
  Function GetShowForm: Boolean;
  Function GetShowSource: Boolean;
  Function NewFormFile(Const FormIdent,
                       AncestorIdent: String): IOTAFile;
  Function NewImplSource(Const ModuleIdent, FormIdent,
                         AncestorIdent: String): IOTAFile;
  Function NewIntfSource(Const ModuleIdent, FormIdent,
                         AncestorIdent: String): IOTAFile;
  Procedure FormCreated(Const FormEditor: IOTAFormEditor);
end;
```

Most of the methods in this class are simply meant to tell the IDE about the various properties of the module that is created. The IDE calls them one by one to get information about the new module.

Some of the more important methods include:

GetExisting Tells the IDE whether this is an existing item or not.

GetOwner Tells the IDE to which module the new file belongs, normally this is the project.

GetUnNamed Tells the IDE whether the file must still be named prior to saving or not.

GetAncestorName Returns the name of the ancestor form name (this is our custom form class).

GetImplFileName Return the name of the source file for the form.

GetFormName Return the new form name.

GetMainForm Determines whether this module is the main form or not.

GetShowForm Determines whether the form must be shown or not in the IDE.

GetShowSource Determines whether the form's source must be shown or not in the IDE.

NewIntfSource Called in C++ Builder to create the header file for the form.

FormCreated Is called by the IDE when the actual form instance was created in the IDE, and it is being edited.

The actual form unit file is created when the IDE calls `NewImplSource`:

```
Function TMKFormWizardModuleCreator.NewImplSource(
   Const ModuleIdent,
   FormIdent,
   AncestorIdent: String): IOTAFile;
Begin
  Result := TMKSourceFile.Create(ModuleIdent,
                                 FormIdent,
                                 AncestorIdent,
                                 FClause) As IOTAFile
```

The `TMKSourceFile` class implements the `IOTAFile` interface and must return the sources (pascal code) for the new, empty form.

Similarly, the actual form file (.dfm) is created when the IDE calls `NewFormFile`:

```
Function TMKFormWizardModuleCreator.NewFormFile(
     Const FormIdent, AncestorIdent: String): IOTAFile;
Begin
  Result:=TMKFormWizardFile.Create(FormIdent,
                                   AncestorIdent,
                                   FUseInherited) As IOTAFile
end;
```

Both the `TMKSourceFile` and `TMKFormWizardFile` classes use the `IOTAFile` to create the form file or unit source for the new form. All new sources in the IDE are created with the `IOTAFile` interface, which looks as follows:

```
IOTAFile = interface(IUnknown) ['{6E2AD9B0-F7F0-11D1-AB26-00C04FB16FB3}']
  function GetSource: string;
  function GetAge: TDateTime; // -1 if new.
  property Source: string read GetSource;
  property Age: TDateTime read GetAge;
end;
```

The properties and methods speak for themselves.

Here is the code which creates the unit sources. It should simply create a string which contains the sources:

```
Constructor TMKSourceFile.Create(Const ModuleIdent,
                                 FormWizardIdent,
                                 AncestorIdent,
                                 UsesClause: String);
Const
  CRLF = #13#10;
  DCRLF = CRLF+CRLF;
begin
  Inherited Create;
  FSource := TStringList.Create;
  With FSource Do
    Begin
    Add(Format('Unit %s;'+DCRLF, [ModuleIdent]));
    Add(Format('Interface'+DCRLF+
               'Uses'+CRLF+
               ' System.SysUtils, System.Classes, WinAPI.Windows,'+
               ' WinAPI.Messages, VCL.Graphics, VCL.Controls,'+
               ' VCL.Forms, VCL.Dialogs, %s;'+DCRLF,[UsesClause]));
    Add(Format('Type'+CRLF+
               '  T%s = Class(T%s)',[FormWizardIdent, AncestorIdent]));
    Add('  Private'+CRLF+
        '    { private declarations }'+CRLF+
        '  Public'+CRLF+
        '    { public declarations }'+CRLF+
        '  end;'+DCRLF);
    Add(Format('Var'+CRLF+
               '  %s: T%s;'+DCRLF,[FormWizardIdent, FormWizardIdent]));
    Add('Implementation'+DCRLF+
        '{$R *.DFM}'+DCRLF);
    Add('End.'+CRLF);
    end;
```

```
end;
```

The sources are returned as created in the - aptly named - `GetSource` method:

```
Function TMKSourceFile.GetSource: String;
Begin
  Result:=FSource.Text;
End;
```

Similarly, the form file can be created:

```
constructor TMKFormWizardFile.Create(
        Const FormWizardIdent, AncestorIdent: String;
        UseInherited: Boolean);
begin
  Inherited Create;
  FSource := TStringList.Create;
  With FSource Do
    Begin
    If UseInherited then
      Add('inherited '+FormWizardIdent+': T'+FormWizardIdent)
    else
      Add('object '+FormWizardIdent+': T'+FormWizardIdent);
    Add(' Left = 192');
    Add(' Top = 103');
    Add(' Width = 640');
    Add(' Height = 480');
    Add('end');
    End;
end;
```

6 Using the new API

The `xptFormWizards` unit can be put in a design-time package (called IDEFormExpert) and must be installed in the IDE, then it is ready for use: it does not install any forms by itself, it simply makes it easier to register custom forms in the IDE.

To demonstrate the new API, we create a simple `TCustomForm` descendant, implemented in `frmAppCustomForm`:

```
TAppCustomForm = Class (TCustomForm)
Public
  Constructor Create(AOwner : TComponent); override;
  Destructor Destroy; Override;
  function  CloseQuery: Boolean; override;
  Procedure Print (ReportName : String);Virtual;
  procedure Preview(ReportName: String);Virtual;
  Procedure RestoreFormSettings;
  Procedure SaveFormSettings;
Published
  // Minimal property set, needed by the IDE.
  Property Caption;
  property PixelsPerInch;
  Property Scaled;
  Property BeforePrint : TBeforePrintEvent;
  Property AfterPrint : TAfterPrintEvent;
  Property OnGetPrintParams : TGetPrintParamsEvent;
  Property OnGetPrintFileName : TGetReportFileNameEvent;
  Property Reports : TStrings;
  Property ReportParams : TStrings;
  Property OnSaveSettings : TSettingsEvent;
  Property OnRestoreSettings : TSettingsEvent;
end;
```

It is a sample class, which implements some of the items mentioned in the introduction of this article, such as saving form position (and possibly other settings), implementing print functionality. The interested reader can consult the sources accompanying this article for the full source code.

This unit, plus some other simple descendants of `TForm`, `TFrame` and `TDatamodule` are implemented in a series of units that are put in a run-time package: AppForms. To be able to use these various forms and descendants, they must be registered in the IDE. That is where the IDEFormExpert package and the `xptFormWizards` unit come in. A design-time package AppFormsIDE is created, which depends on the 2 packages: IDEFormExpert and AppForms. The design-time package contains a single unit: RegAppForms, which registers the various forms in the IDE:

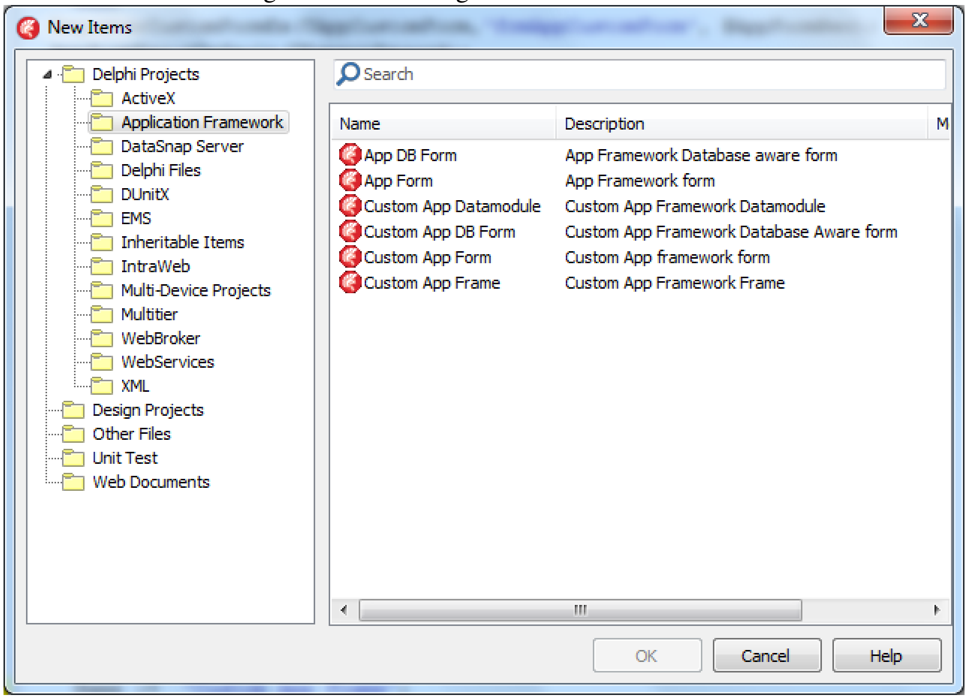
```
Procedure Register;

Const
  AppPage = 'Application Framework';

Var
  AppFormRec : TExpertRecord;

begin
  RegisterComponents('Application Framework',[TAppDataset]);
  AppFormRec:=Default(TExpertRecord);
  With AppFormRec do
    begin
    Name :=  'Custom App Form';
    Author := 'Michael Van Canneyt';
    Comment := 'Custom App framework form';
    Page := AppPage;
    Glyph := 0;
    IDString := 'App.CustomAppForm.Form';
    end;
  RegisterCustomFormEx(TAppCustomForm,
                       'frmAppCustomForm', @AppFormRec);
  AppFormRec:=Default(TExpertRecord);
  With AppFormRec do
    begin
    Name :=  'Custom App DB Form';
    Author := 'Michael Van Canneyt';
    Comment := 'Custom App Framework Database Aware form';
    Page := AppPage;
    Glyph := 0;
    IDString := 'App.CustomAppForm.DBForm';
```

Figure 2: Custom registered forms in the IDE



Name	Description
App DB Form	App Framework Database aware form
App Form	App Framework form
Custom App Datamodule	Custom App Framework Datamodule
Custom App DB Form	Custom App Framework Database Aware form
Custom App Form	Custom App framework form
Custom App Frame	Custom App Framework Frame

```
  end;
RegisterCustomFormEx(TDBAppCustomForm,
                     'frmDBAppCustomForm', @AppFormRec);
```

More forms and frames are registered in this manner. The package AppFormsIDE must be installed in the IDE. When doing so, and the 'File-New-other' menu item is chosen, then the image in figure 2 on page 11 is visible... The sources for the forms plus a demo application that shows how to use them, are available together with the article.

7 Conclusion

Given the auxiliary routines presented here (in the xptFormWizards unit), registering your own form in the IDE is not very hard. The methods presented here create static sources: the content is always the same (save the class names for the form) and can be easily extended to present a dialog that can be used to modify the generated sources and form file. A problem that is not solved yet is the fact that the functionality of Visual Form Inheritance cannot be duplicated. This must be worked around, but this is left for a future contribution.
Acknowledgement: An embryonic form of the code presented here was created (a very long time ago) by Franck Musson.