The Delphi Open Tools API

1. Forward

Well I've never written a book before so this may not be War and Peace on the *Open Tools API* that everyone wants but I hope that it goes some way towards filling the gap in the documentation that has existed. I would, if I had time, like to cover all of the *Open Tools API*, especially the `ToolsAPI.pas` file but that is a massive undertaking which I don't have the time to do unfortunately. Should I get a nice 9 to 5 job around the corner from where I live which would keep me in the life I'm accustom to, then I might have a go.

For me working with the *Open Tools API* started with Delphi 3 all those many moons ago. The *Open Tools API* started with Delphi 2 (first 32-bit version) which allowed you to programme the IDE however it used a very different interface mechanism than is not in use today and one that I cannot even remember. I got a little help back then from an excellent book by *Ray Lischner* called the *Hidden Paths of Delphi 3* which I read from start to finish (and still have) and if I remember correctly, created all the example code contain there in. I still think that if you can find this book its worth reading. There are many aspects to the *Open Tools API* and this book generally only covers those aspect contained within the `ToolsAPI.pas` file. There are many other files in the Open Tools API (have a look at the `Source\ToolsAPI\` directory in you installation, these files allow you to do other things like create your own property inspectors, virtual filing systems, proxies and much more.

Why did I start to investigate the *Open Tools API*? Well initially it seemed an interesting thing to learn (back in the days when I have way too much time on my hands) but eventually there came a point when I needed to solve a problem that the IDE either didn't do or didn't do it properly. The first instance of this with Delphi 5's module/code explorer which would just lock up (the rest of the IDE functioned). So I wrote *PascalDoc*, a code browser and documentation tool. This has since been supersede by *Browse and Doc It* which does the browsing, the documentation but also helps with profiling code with instrumentation and creating *DUnit* projects and modules. Later on when I started to use *DUnit* more and wanted to automate the compiling and running of the tests before compiling the main project, I created the *Integrated Testing Helper*.

The contents of this book are mostly a collection of blogs I wrote over a number of years on the Delphi Open Tools API. I haven't done much to the chapters other than correct spelling mistakes and grammar as the chapters describe a journey through a learning process which I hope the reader will appreciate.

The chapter numbers are different here than the original blogs as I've included all the incidental information on the Open Tools API which I think adds to this knowledge base. There are a number of mistakes in some of the articles which are identified and corrected in later chapters as they were in the original blogs. It is therefore suggested to read the whole of this book to ensure you know about any issues in the earlier articles.

The reason for the blogs originally was to write down all that I had found out while trying to implement *Open Tools API* code so that I had a reference as to why I had done things in a particular way but also to provide the same information to the wider Delphi community. I've also written a number of new chapters on topics that have been outstanding for several years.

Additionally, all the referenced code examples are contained in a number of ZIP files attached to this PDF as attachments. The example code contained in the PDF is free for all to use as they see fit. The code for *Browse and Doc It* and the *Integrated Testing Helper* is provided for reference. The *Open Tools API* aspects of the code are also free to be used however I do reserve all writes to the remaining code which is provide for information purposes only and not for redistribution or reuse.

I haven't done very much with Delphi and the *Open Tools API* in the past few years other than maintain some of y existing applications however I'm not down and out yet! Although I don't know whether I'll get time to do any more investigation into the *Open Tools API* I have in the back of my mind that I would update this book and publish the additional material for all. In trying to finish this book I skipped across an `IOTA` interface for the help system. One of the annoyances of the latest IDEs

is their lack of MS Win32/64 SDK help so I might investigate whether I can intercept this F1 context key press and redirect them to a browser and bring up the MSDN reference. Like most of these things I have done, I have no idea whether I can do it but I'll have a go an let you know.

Finally, I'm sure that there are going to be some spelling mistakes, grammatical errors as well as issues with the code so I would appreciate constructive feedback on the book, its contents, style, etc. and I will attempt to update the book at a future time. Also I would like to thank David Millington for encouraging me to get my head backing into the Open Tools API and writing this book.

Hope you all enjoy.

Regards

David Hoyle @ Feb 2016

Website: http://www.davidghoyle.co.uk/
Email: davidghoyle@gmail.com
Blog: http://www.davidghoyle.co.uk/WordPress/
Music: http://www.davidghoyle.co.uk/Music

2. Contents

3. Starting an Open Tools API Project

This article was originally published on 07 Aug 2009 using RAD Studio 2006.

3.1 Before You Start

3.1.1 Think about your audience

Before you start, it's a good idea to think a little about the management of the project files. Although you may think that you'll build this add-in for your current version of the IDE, if you upgrade and require backward compatibility (or choose to distribute your add-in) you may come to regret some of your previous decisions. So below are some suggestions about the organisation of information and management of files based around the way I've found most flexible.

3.1.2 Structure

I like to structure my project directory to keep different types of files in different places. The directories I create for an IDE Add-in projects are as follows:

- `DCUs` – for the DCU units output from the project;
- `DLL` – for the DLL source (.dpr) and associated project files;
- `Package` – for the BPL source (.dpk) and associated project files;
- `Source` – for the forms and units that make up the project.

The DCUs and Source directories should be self-explanatory. The DLL and package directories are for the DLL and Package version of the project we're going to build. This just keeps the project root directory clearer.

3.1.3 Name of Projects

Append to the end of the project name the IDE version, i.e. 50 for Delphi 5 or 2006 for BDS 2006. Always create a new set of project files for each new compiler, you cannot maintain different versions though the same project files. I've noticed that RAD Studio 2009 (don't have 2007) doesn't allow 2 project with the same name like BDS 2006 does in the project group, so you may have to append an extra P to the end of the package version.

3.2 Bare Bones

Open your IDE, I'm going to be using BDS 2006, but all of what I will do can be done with Delphi 5 through to RAD Studio 2009. Do the following:

- Add a new project group;
- Right click on the project group and add a new DLL project;
- Right click again on the project group and add a new Package project.
- Next select “Save All” from the File menu and save the DLL project to the `DLL` directory, the Package project to the `Package` directory and save the project group to the project root directory.

Next we need to configure the project options for each of the two projects. Fill in the options as you see fit, but there are 2 that need specific attention due to the directory structure we're using. These are *Output Directory* and *Unit Output Directory* and should be configured as shown the in image below.

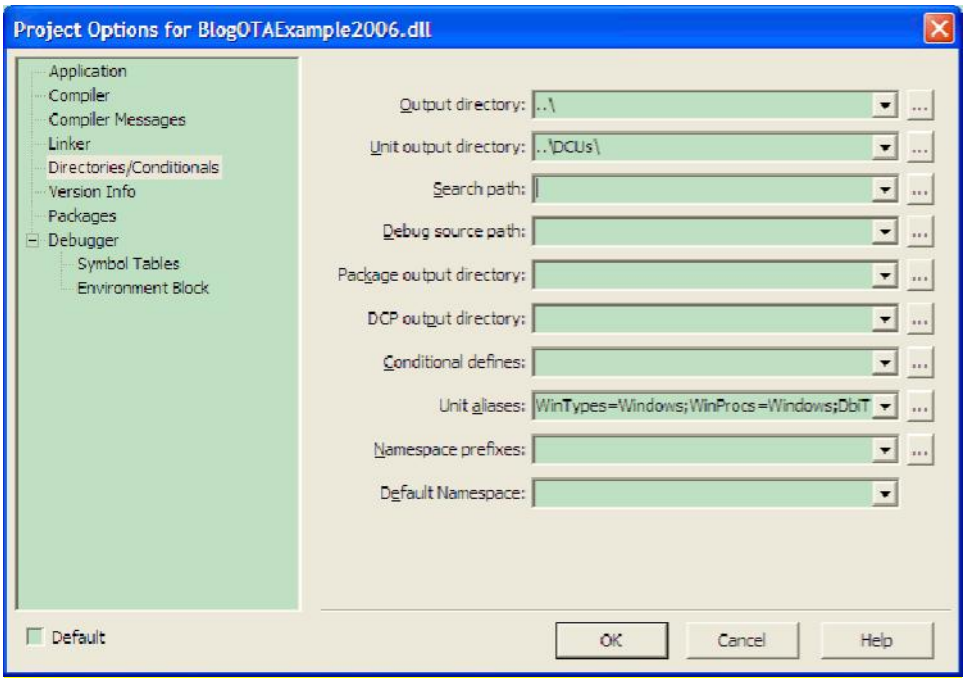


Figure 1: OTA Project Options

There is one further change that needs to be made to both projects in order that they can access the IDEs Open Tools API interfaces and they are handled slightly different for DLLs and Packages. For the DLL you need to add `DesignIDE` to the list of packages in the `.dpr` file but for the Package you need to add `DesignIDE` to the packages `Requires` clause in the `.dpk` file.

At this point the projects will compile but they will not actually do anything if loaded by the IDE. So the next step is to create a simple IDE expert / wizard.

3.3 Creating a Simple Wizard

The code which follows (and has come before) can be used as a template to all new wizards. The differences between the wizards depend on the code you write inside the wizard and the interfaces that your wizard implements.

We need to start with a new unit, so right click on the DLL project and select a new unit. Save this to the `Source` directory with a meaningful name like `BlogOTAExampleWizard.pas`.

Next we need to add the Wizard class definition as follows:

```
Unit BlogOTAExampleWizard;

Interface

Uses
  ToolsAPI;

Type
  TBlogOTAExampleWizard = Class(TInterfaceObject, IOTAWizard)
  End;

End.
```

At this point we need to implement the methods of `IOTAWizard` which aren't already implemented by our ancestor class `TInterfacedObject`. In BDS 2006 and above there's an easy way – place the cursor at the start of the End keyword in the class definition and press `Ctrl+Space`.

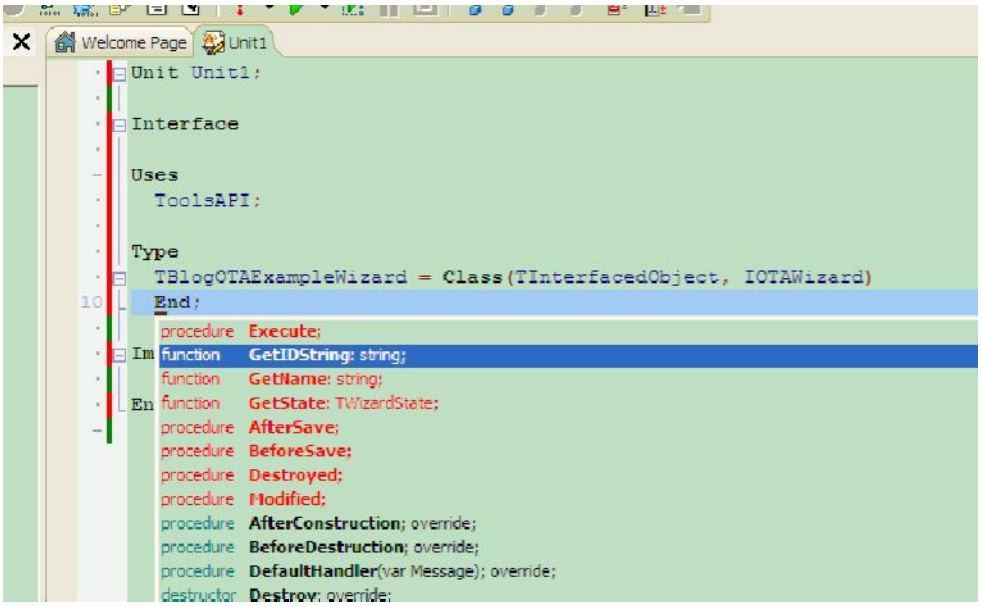


Figure 2: Ctrl+Space in a class to create the missing interface methods

The methods that need to be implemented are the one in red. If you select all the methods highlighted red (use the shift key and down arrow) and press <Enter>, then those methods will be added to the class declaration.

```
Unit Unit1;

Interface

Uses
  ToolsAPI;

Type
  TBlogOTAExampleWizard = Class(TInterfaceObject, IOTAWizard)
  Public
    Procedure Execute;
    Function  GetIDString : String;
    Function  GetName : String;
    Function  GetState : TWizardState;
    Procedure AfterSave;
    Procedure BeforeSave;
    Procedure Destroyed;
    Procedure Modified;
  End;

Implementation

End.
```

Use class completion (Ctrl+C) to write the implementation declarations of these methods and save the unit as BlogOTAExampleWizard.pas in the Source directory if you haven't already done so. To ensure that the Package also uses this same unit, drag and drop the unit from the DLL to the Package.

3.4 Implementing the Interface Methods

Now we need to implement the interface's methods that weren't handled by TInterfacedObject, i.e. the methods created by Class Completion above. These methods should be:

- Procedure Execute;
- Function GetIDString: String;
- Function GetName: String;
- Function GetState: TWizardState;
- Procedure AfterSave; // Not called for IOTAWizards;

- `Procedure BeforeSave; // Not called for IOTAWizards;`
- `Procedure Destroyed;`
- `procedure Modified; // Not called for IOTAWizards.`

So from above we only need to implement 5 out of the 8 methods as the other 3 are not called in wizards.

We only need to implement the `Destroyed` method IF we need to know when the wizard is being destroyed so that we can free memory used by the wizard. In this example we have no need to implement `Destroyed`.

3.4.1 GetIDString

This method returns to the IDE a unique identification string to distinguish your add-in from all others. Combine your name / company with the name of the add-in.

```
Function TBlogOTAExampleWizard.GetIDString : String;

Begin
  Result := 'David Hoyle.BlogIOTAExample';
End;
```

3.4.2 GetName

This method returns to the IDE a name for your add-in.

```
Function TBlogOTAExampleWizard.GetName : String;

Begin
  Result := 'Open Tools API Example';
End;
```

3.4.3 GetState

This method returns the state of the add-in. This is a set of enumerates which only contains 2 items – `wsEnabled` and `wsChecked`. For our add-in we'll return just `wsEnabled`.

```
Function TBlogOTAExampleWizard.GetState : TWizardState;

Begin
  Result := [swEnabled];
End;
```

3.4.4 Execute

This method for the moment will be left empty.

3.5 Making the IDE See the Wizard

At the moment our add-in will compile and could be loaded into the IDE (see below) but the IDE will complain that it can't find the wizard. DLLs and Packages handle the informing of the IDE differently BUT you can still use the same code.

3.5.1 DLLs

DLLs require a function named `InitWizard` with the following signature:

```
Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
  RegisterProc : TWizardRegisterProc;
  Var Terminate : TWizardTerminateProc) : Boolean; StdCall;

Begin
  InitialiseWizard;
End;
```

Don't worry about the `InitialiseWizard` call just yet; we'll come to that in a minute. This function also needs to be exported by the DLL so that the IDE can know that the function exists in the DLL and

initialise the DLL. To export the function you need to add the following declaration to your wizard unit including the interface definition of the `InitWizard` function.

```
  Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
    RegisteProc : TWizardRegisterProc;
    Var Terminate : TWizardTerminateProc) : Boolean; StdCall;

Exports
  InitWizard Name WizardEntryPoint;
```

3.5.2 Packages

Packages do thing slightly differently by requiring a procedure call `Register`. *Note this is case sensitive*.

```
Procedure Register;

Begin
  InitialiseWizard;
End;
```

Again, ignore the `InitialiseWizard` call. As with the DLL above we need to declare the interface declaration of the Register procedure.

3.5.3 InitialiseWizard

I've always started in the past by building my add-in as packages and then later on making them DLL compatible. This has led to duplication of code in the 2 different initialisation procedures so I'm going to try a different approach and use a single initialisation procedure to do all the work – this is work in progress so bear with me here :-)

DLLs and Packages load their wizards differently through different mechanism but we would like to minimise the duplication of code as much as possible. DLLs load the wizard by passing an instance of the wizard to the procedure `RegisterProc` in the signature of `InitWizard`. Packages on the other hand use the `IOTAWizardServices` to add the wizard to the IDE. The reason for the differences are to do with the fact that Packages can be loaded and unloaded dynamically throughout the life time of the IDE but DLLs are loaded only once during the IDE start-up process. Below is the code needed to initialise the wizard in the different mechanisms.

```
Var
  iWizard : Integer = 0;

Function InitialiseWizard(BIDES : IBorlandIDEServices) : TBlogOTAExampleWizard;

Begin
  Result := TBlogOTAExampleWizard.Create;
  Application.Handle := (BIDES As IOTAServices).GetParentHandle;
End;

Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
  RegisterProc : TWizardRegisterProc;
  Var Terminate : TWizardTerminateProc) : Boolean; StdCall;

Begin
  Result := BorlandIDEServices <> Nil;
  RegisterProc(InitialiseWizard(BorlandIDEServices));
End;

Procedure Register;

Begin
  iWizard := (BorlandIDEServices As IOTAWizardServices).AddWizard(
    InitialiseWizard(BorlandIDEServices));
End;
```

You will notice that `InitialiseWizard` is a function which returns the instance of our wizard to the 2 different methods for loading the wizard in DLLs and Packages. With the DLL, once the wizard is created it does not need freeing but the Package version does. You will notice that `AddWizard` returns an integer reference for the wizard. This is used in a call to `RemoveWizard` to remove the wizard from memory. The best location for this is in the units `Finalization` section as below.

```
Initialization
Finalization
  If iWizard > 0 Then
    (BorlandIDEServices As IOTAWizardServices).RemoveWizard(iWizard);
End.
```

You will notice a couple of things about these 2 pieces of code. Firstly, the `iWizard` integer is initialised to zero – this is done so that the call to `RemoveWizard` is only used if the `iWizard` variable is greater than zero. `iWizard` will remain zero for a DLL and therefore the `RemoveWizard` call will not be made for DLLs only Packages.

Another thing you might have noticed is that there are no calls to free the wizard. This is because this is done by the IDE for you. If you are not convinced, simply add a constructor and destructor to the wizard and add `OutputDebugString` calls to each and debug the application (see below).

3.6 Telling the IDE about the Wizard

To get the IDE to load the wizard you need to do 2 different things for DLLs and Packages. For packages simply load the IDE, open the package project, right click on the package and select "Install" and the package will be compiled and installed into the IDE. For a DLL it's a little more complicated. We need to add a new register key to the IDE's register entries. Since I'm using BDS 2006. I'll use its registry as an example but you will see it's easy to do the same for other versions of the IDE. BDS 2006 stores its information under the registry location "`My Computer\HKEY_CURRENT_USER\Software\Borland\BDS\4.0\`". We need to add a new key (if not already there) called "`Experts`". Inside this key we need to add a new string entry named after the add-in with its value being the `drive:\path\filename.ext` of the DLL as below.

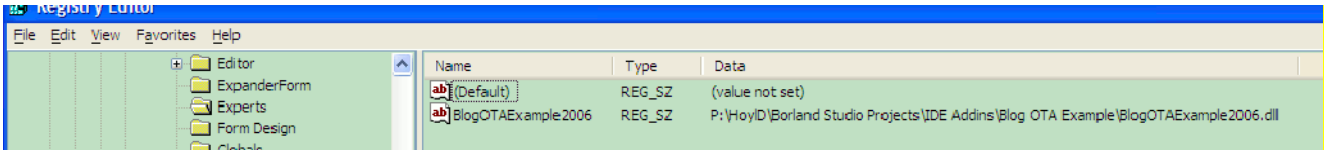


Figure 3: A screen shot of the registry editor showing the new key and value.

Now we can run the add-in wizard as either a package or DLL – *DON'T DO BOTH*!

3.7 Testing and Debugging

You could get instant gratification from your new add-in wizard by installing the package but if you've got things wrong then you could crash the IDE so it's better to test and debug your add-in in a second instance of the IDE first. According to the Delphi Wiki entry there has been a command line switch for the IDE since Delphi 7 that tells it to use a different register key. I've never tried it with anything earlier than BDS 2006 but this works well. You need to setup the parameters for the debugging as below (note, use a different key for DLLs and Packages so that you don't get them both loading). Additionally, for these secondary IDEs the above `Experts` keys will need to be made in the alternate registry location (instead of `\BDS\4.0\ it will be \OpenToolsAPI\4.0\`).

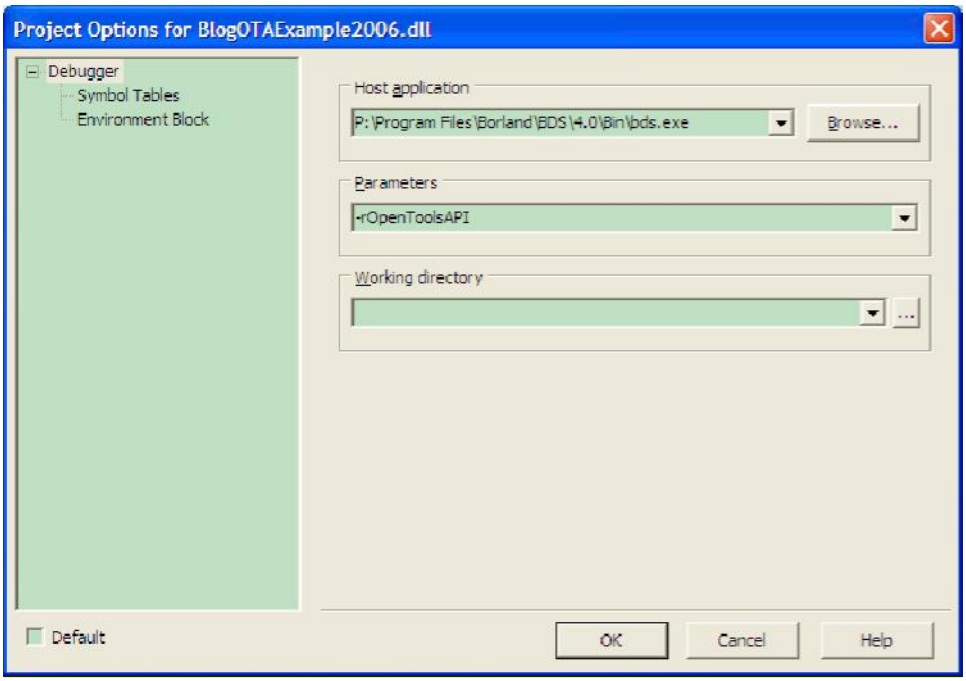


Figure 4: A screen shot of the debugging options with settings for debugging the add-in.

Now if you run the DLL and debug it you should find that it does exactly nothing BUT doesn't crash the IDE on loading or unloading. Now we need to make the wizard do something.

3.8 Making the Wizard Actual Do Something

To make the add-in do something we need to add another interface to our class `IOTAMenuWizard` and implement 1 new method fucntion `GetMenuText`. Do not changes the current `IOTAWizard` reference to `IOTAMenuWizard` as your code will not compile. The `IOTAWizard` interface is required by `RegisterProc`. In the new method, just return a string representing the text of your menu item. This menu item will appear under the Help menu. There is one more thing to do and that is do something in the `Execute` method. Add a call to `ShowMessage` with a simple string like `"Hello World!"` – you will have to add the `Dialogs` unit to the uses clause for the add-in to compile.

If you run the add-in now and select the menu item you will get your message displayed.

3.9 Conclusion

I hope this has been straight forward. In the next chapter we'll provide a proper menu interface, one that appears within the IDEs menus :-)

The source files for this example are attached to this PDF as `OTAChapter03StartingAnOpenToolsAPIProject.zip`.

4. A simple custom menu (AutoSave)

This was originally published on 13 Aug 2009 using RAD Studio 2006.

I thought that this time I would give you something useful for a change, so while implementing a simple custom menu we'll create a wizard that provides an auto save feature for the IDE.

With this chapter I'm changing tact and instead of screen shots I'll include the source code (syntax highlighted). This means you can copy and paste the code easily. There will still be a zip file at the end of each chapter for the entire source code to the project.

4.1 Creating a Form Interface for the Wizard

First we need to do some groundwork to create a form for editing the auto save options. I'm assuming here that you are familiar with forms so I'm just doing to give you the code, first the `.pas` file and then the `.dfm` so you can paste the information into your IDE and we can get on with the interesting stuff. Either using the projects we created last time or copies of those projects add a new form to the DLL project and replace all the code with the following (REMEMEBER: if you create a copy of the projects, change the `GetIDString` and `GetName` methods to reflect a different wizard):

```
unit OptionsForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons, ComCtrls;

type
  TfrmOptions = class(TForm)
    lblAutoSaveInterval: TLabel;
    edtAutosaveInterval: TEdit;
    udAutoSaveInterval: TUpDown;
    cbxPrompt: TCheckBox;
    btnOK: TBitBtn;
    btnCancel: TBitBtn;
  private
    { Private declarations }
  public
    { Public declarations }
    Class Function Execute(var iInterval : Integer; var boolPrompt : Boolean) :
       Boolean;
  end;

implementation

{$R *.DFM}

{ TfrmAutoSaveOptions }

class Function TfrmOptions.Execute(var iInterval: Integer;
  var boolPrompt: Boolean) : Boolean;

begin
  Result := False;
  With TfrmOptions.Create(Nil) Do
    Try
      udAutoSaveInterval.Position := iInterval;
      cbxPrompt.Checked := boolPrompt;
      If ShowModal = mrOK Then
        Begin
          Result := True;
          iInterval := udAutoSaveInterval.Position;
          boolPrompt := cbxPrompt.Checked;
        End;
    Finally
```

```
      Free;
    End;
end;

end.
```

Next view the form and right click on the form and selected “View as Text” and replace all the code with the following:

```
object frmOptions: TfrmOptions
  Left = 443
  Top = 427
  BorderStyle = bsDialog
  Caption = 'Auto Save Options'
  ClientHeight = 64
  ClientWidth = 241
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  Position = poScreenCenter
  PixelsPerInch = 96
  TextHeight = 13
  object lblAutoSaveInterval: TLabel
    Left = 8
    Top = 12
    Width = 88
    Height = 13
    Caption = 'Auto Save &Interval'
    FocusControl = edtAutosaveInterval
  end
  object edtAutosaveInterval: TEdit
    Left = 104
    Top = 8
    Width = 41
    Height = 21
    TabOrder = 0
    Text = '60'
  end
  object udAutoSaveInterval: TUpDown
    Left = 145
    Top = 8
    Width = 15
    Height = 21
    Associate = edtAutosaveInterval
    Min = 60
    Max = 3600
    Position = 60
    TabOrder = 1
  end
  object cbxPrompt: TCheckBox
    Left = 8
    Top = 36
    Width = 97
    Height = 17
    Caption = '&Prompt'
    TabOrder = 2
  end
  object btnOK: TBitBtn
    Left = 164
    Top = 8
    Width = 75
    Height = 25
    TabOrder = 3
    Kind = bkOK
  end
```

```
  object btnCancel: TBitBtn
    Left = 164
    Top = 36
    Width = 75
    Height = 25
    TabOrder = 4
    Kind = bkCancel
  end
end
```

Now save the form in the Source directory as `OptionsForm.pas`.

4.2 Updating the Wizard

We can now get on with the fun bits of this chapter. First we need to update the class declaration of the wizard – we will not be needing the `IOTAMenuWizard` interface so this can be removed and the `GetMenuText()` method deleted.

```
  TBlogOTAExampleWizard = Class(TInterfacedObject, IOTAWizard)
  Public
    FTimer       : TTimer;                  // New
    FCounter     : Integer;                 // New
    FAutoSaveInt : Integer;                 // New
    FPrompt      : Boolean;                 // New
    FMenuItem    : TMenuItem;               // New
    FINIFileName : String;                  // New
    Procedure SaveModifiedFiles;            // New
  Protected
    procedure Execute;
    function  GetIDString: string;
    function  GetName: string;
    function  GetState: TWizardState;
    procedure AfterSave;
    procedure BeforeSave;
    procedure Destroyed;
    procedure Modified;
    // function  GetMenuText: string;       // Deleted
    Procedure TimerEvent(Sender : TObject); // New
    Procedure MenuClick(Sender : TObject);  // New
    Procedure LoadSettings;                 // New
    Procedure SaveSettings;                 // New
  Public
    Constructor Create;                     // New
    Destructor Destroy; Override;           // New
  End;
```

Next we need to update the uses clause in the implementation section to provide access to other modules that will be required. I've taken this opportunity to rename the wizard index variable so that it's clear what it refers to:

```
Uses
  // Delete Dialogs and add Windows, SysUtils, OptionsForm and IniFiles
  Forms, Windows, SysUtils, OptionsForm, IniFiles;

Var
  (** A private variable to hold the index returned by AddWizard which is needed
      by RemoveWizard. **)
  iWizardIndex : Integer = 0; // Renamed
```

Next we need to code the constructor. We need to initialise our fields, start the timer and create the menu as follows:

```
constructor TBlogOTAExampleWizard.Create;

var
  NTAS: INTAServices;
  mmiViewMenu: TMenuItem;
  mmiFirstDivider: TMenuItem;
```

```
  iSize : DWORD;

begin
  FMenuItem := Nil;
  FCounter := 0;
  FAutoSaveInt := 300; // Default 300 seconds (5 minutes)
  FPrompt := True;     // Default to True
  // Create INI file same as add module + '.INI'
  SetLength(FINIFileName, MAX_PATH);
  iSize := MAX_PATH;
  iSize := GetModuleFileName(hInstance, PChar(FINIFileName), iSize);
  SetLength(FINIFileName, iSize);
  FINIFileName := ChangeFileExt(FINIFileName, '.INI');
  LoadSettings;
  FTimer := TTimer.Create(Nil);
  FTimer.Interval := 1000; // 1 second
  FTimer.OnTimer := TimerEvent;
  FTimer.Enabled := True;
  NTAS := (BorlandIDEServices As INTAServices);
  If (NTAS <> Nil) And (NTAS.MainMenu <> Nil) Then
    Begin
      mmiViewMenu := NTAS.MainMenu.Items.Find('View');
      If mmiViewMenu <> Nil Then
        Begin
          //: @bug Menu not fully build at this point.
          mmiFirstDivider := mmiViewMenu.Find('-');
          If mmiFirstDivider <> Nil Then
            Begin
              FMenuItem := TMenuItem.Create(mmiViewMenu);
              FMenuItem.Caption := '&Auto Save Options...';
              FMenuItem.OnClick := MenuClick;
              mmiViewMenu.Insert(mmiFirstDivider.MenuIndex, FMenuItem);
            End;
        End;
    End;
end;
```

You will note that I've marked the menu creation code with a bug comment. What I found was that loading this wizard as a DLL loads the code much earlier in the IDE start-up process than loading it as a package. The consequence of this is that not all the IDE menus have been fully loaded. Originally, I was looking for the “Window List” menu item and inserting this new menu below it. I've copped out here and found the first separator in the menu and inserted the new menu above it. I will address this problem along with keyboard short cuts for menus in the next instalment. This only affects finding an IDE menu to reference against – creating your own top level menu would not be affected. I'll do this in a later chapter.

There's something else of interest in this code as well. I gave up using the windows registry some time ago as it can't be backed up such that you can restore your settings – so I elected to move back to the old fashioned INI file. Although I use slightly different code in my own applications (placing the users name and computer name in the INI file name) this is essentially what I do. I use the Win32 API `GetModuleFileName` and pass it the `hInstance` global variable. What this means is that for DLLs and BPLs I get the name of the DLL, but for EXE I get the EXE name. If you were to use `ParamStr(0)` in the IDE you would get the Delphi / RAD Studio EXE name.

4.3 Loading and Saving the Settings

Next we need to code the destructor to ensure we free all the memory we've used as follows:

```
destructor TBlogOTAExampleWizard.Destroy;
begin
  SaveSettings;
  FMenuItem.Free;
  FTimer.Free;
  Inherited Destroy;
end;
```

You will notice that I call `FMenuItem.Free` even though it might not have been initialised (i.e. if the menu position was not found). This is in fact absolutely fine. `Free` is a class method and therefore can be called on a `NIL` reference, hence why I ensure it's initialised to `NIL` in the constructor. One of the Borland / CodeGear guys wrote about this a couple of years ago and explain why this would work – I just don't think it's widely known.

The next thing to do is implement the loading and saving code for the wizard's settings as follows:

```
procedure TBlogOTAExampleWizard.LoadSettings;

begin
  With TIniFile.Create(FINIFileName) Do
    Try
      FAutoSaveInt := ReadInteger('Setup', 'AutoSaveInt', FAutoSaveInt);
      FPrompt := ReadBool('Setup', 'Prompt', FPrompt);
    Finally
      Free;
    End;
end;

procedure TBlogOTAExampleWizard.SaveSettings;

begin
  With TIniFile.Create(FINIFileName) Do
    Try
      WriteInteger('Setup', 'AutoSaveInt', FAutoSaveInt);
      WriteBool('Setup', 'Prompt', FPrompt);
    Finally
      Free;
    End;
end;
```

We can expand these routines later to load and save more settings.

4.4 Detecting the modified files and saving them

Next we'll code the timer event handler. We simply call the `SaveModifiedFiles` routine when the counter gets larger than the interval and reset the counter at the same time.

```
procedure TBlogOTAExampleWizard.TimerEvent(Sender: TObject);

begin
  Inc(FCounter);
  If FCounter >= FAutoSaveInt Then
    Begin
      FCounter := 0;
      SaveModifiedFiles;
    End;
end;
```

Next we'll code the `MenuClick` event handler passing our two fields as parameters so that the options dialogue at the start of this chapter can modify the values.

```
procedure TBlogOTAExampleWizard.MenuClick(Sender: TObject);

begin
  If TfrmOptions.Execute(FAutoSaveInt, FPrompt) Then
    SaveSettings; // Not really required as is called in destructor.
end;
```

Finally we come to the interesting bit, saving the modified files in the IDE.

```
procedure TBlogOTAExampleWizard.SaveModifiedFiles;

Var
  Iterator : IOTAEditBufferIterator;
  i : Integer;
```

```
begin
  If (BorlandIDEServices As IOTAEditorServices).GetEditBufferIterator(Iterator) Then
    Begin
      For i := 0 To Iterator.Count - 1 Do
        If Iterator.EditBuffers[i].IsModified Then
          Iterator.EditBuffers[i].Module.Save(False, Not FPrompt);
    End;
end;
```

Here we ask the IDE for a Edit Buffer Iterator and use that iterator to check each file in the editor to see if it has been modified and if it has then save the modifications.

Well I hope this is straight forward.

The source for this chapter is attached to this PDF as `OTAChapter04ASimpleCustomMenu(AutoSave).zip`.

Note: *There is a fix to this code in the chapter A simple custom menu (AutoSave) Fixed*.

5. A simple custom menu (AutoSave) Fixed

This was originally published on 20 Sep 2009 using RAD Studio 2006.

In the last chapter I said that the mechanism that I had might not work property with DLLs because if you are looking for an existing menu, then it might not have been created at the time your wizard is loading. So here I'm going to show you how to fix that.

First we need to add a new method to the class which will take the menu creation code out of the classes `Create` method:

```
  TBlogOTAExampleWizard = Class(TInterfacedObject, IOTAWizard)
  Private
    FTimer       : TTimer;
    FCounter     : Integer;
    FAutoSaveInt : Integer;
    FPrompt      : Boolean;
    FMenuItem    : TMenuItem;
    FINIFileName : String;
    FSucceeded   : Boolean; // New
    Procedure SaveModifiedFiles;
    Procedure LoadSettings;
    Procedure SaveSettings;
    Procedure InstallMenu; // New
    Procedure TimerEvent(Sender : TObject);
    Procedure MenuClick(Sender : TObject);
  Protected
    procedure Execute;
    function  GetIDString: string;
    function  GetName: string;
    function  GetState: TWizardState;
    procedure AfterSave;
    procedure BeforeSave;
    procedure Destroyed;
    procedure Modified;
  Public
    Constructor Create;
    Destructor Destroy; Override;
  End;
```

Next we need to remove the menu code from the `Create` method (cut it to the clipboard as you will need it in a minute):

```
constructor TBlogOTAExampleWizard.Create;

var
  iSize : DWORD;

begin
  FMenuItem := Nil;
  FCounter := 0;
  FAutoSaveInt := 300; // Default 300 seconds (5 minutes)
  FPrompt := True;     // Default to True
  // Create INI file same as add module + '.INI'
  SetLength(FINIFileName, MAX_PATH);
  iSize := MAX_PATH;
  iSize := GetModuleFileName(hInstance, PChar(FINIFileName), iSize);
  SetLength(FINIFileName, iSize);
  FINIFileName := ChangeFileExt(FINIFileName, '.INI');
  LoadSettings;
  FSucceeded := False;
  FTimer := TTimer.Create(Nil);
  FTimer.Interval := 1000; // 1 second
  FTimer.OnTimer := TimerEvent;
  FTimer.Enabled := True;
end;
```

Next we need to add the new method `InstallMenu`. You will note it has changed slightly from before as I wanted the "Auto Save..." menu to appear below the "Window List..." menu item.

```
procedure TBlogOTAExampleWizard.InstallMenu;

Var
  NTAS: INTAServices;
  mmiViewMenu: TMenuItem;
  mmiWindowList: TMenuItem;

begin
  NTAS := (BorlandIDEServices As INTAServices);
  If (NTAS <> Nil) And (NTAS.MainMenu <> Nil) Then
    Begin
      mmiViewMenu := NTAS.MainMenu.Items.Find('View');
      If mmiViewMenu <> Nil Then
        Begin
          mmiWindowList := mmiViewMenu.Find('Window List...');
          If mmiWindowList <> Nil Then
            Begin
              FMenuItem := TMenuItem.Create(mmiViewMenu);
              FMenuItem.Caption := '&Auto Save Options...';
              FMenuItem.OnClick := MenuClick;
              FMenuItem.ShortCut := TextToShortCut('Ctrl+Shift+Alt+A');
              mmiViewMenu.Insert(mmiWindowList.MenuIndex + 1, FMenuItem);
              FSucceeded := True;
            End;
        End;
    End;
end;
```

You should also note from above that I've added a shortcut to the menu item.

You should have noticed by now that there is a new field called `FSuccess` in the class which is initialised in the constructor to `False`. This is to indicate that our new menu has not been installed yet. Next we need to update the `TimeEvent` method to reference this new field and install the menu as follows:

```
procedure TBlogOTAExampleWizard.TimerEvent(Sender: TObject);
begin
  Inc(FCounter);
  If FCounter >= FAutoSaveInt Then
    Begin
      FCounter := 0;
      SaveModifiedFiles;
    End;
  If Not FSucceeded Then
    InstallMenu;
end;
```

That's it for the changes. Hopefully that's straight forward.

The files for this chapter are attached to this PDF as `OTAChapter05ASimpleCustomMenu(AutoSave)Fixed.zip`.

6. IOTA Interfaces...

This was originally published on 25 Oct 2009 using Delphi 7 Professional.

While finalising the release of *Browse and Doc It* I found that the IDE add-in was causing the Delphi 7 IDE to crash, but not all the time. Very frustrating to say the least but I did manage to understand why and there's a lesson to be learnt.

I had introduced 2 new Syntax Highlighters into the IDE with declarations similar to below:

```
TBNFHighlighter = Class(TNotifierObject, IOTAHighlighter {$IFDEF D2005},
      IOTAHighlighterPreview {$ENDIF})
```

Some of you will probably say "You Clot", but I didn't understand until something I had read in the past crept back into my mind. You must implement ALL the interfaces in the inheritance list of the interfaces you require. So what does this actually mean? Well from what I remember of what I read, interfaces do not actual use inheritance! It appears that the IDE expects all the interfaces in the "Chain" of interfaces to be implemented so `IOTAHighlighter` requires an additional interface `IOTANotifier` as follows:

```
TBNFHighlighter = Class(TNotifierObject, IOTANotifier, IOTAHighlighter {$IFDEF
      D2005}, IOTAHighlighterPreview {$ENDIF})
```

Once I had done this the crashes reduced in number (there were other issued to sort out) but there were none at the lines of code to remove the Highlighters from the IDE.

I found another instance of an IDE object I was creating like this and that reduced the errors to almost zero. I would seem that the later IDEs 2006 to 2009 are either more resilient than the Delphi 7 IDE or the way the code is compiled does not show the errors.

Hope this helps people.

BTW I had only found this by debugging the `Destroy` and `Finalization` sections of the code by stepping through them.

7. Key Bindings and Debugging Tools

This was originally published on 10 Feb 2010 using RAD Studio 2009.

In this chapter I'll solve a problem I've found with all the Borland/CodeGear IDE's I've used. I like to use just the keyboard where possible rather than grab the mouse. With the keyboard (and I use the IDE Classic keyboard binding) you can create a breakpoint with `Ctrl+F8` but you cannot edit its properties. You also can't disable the breakpoint with the keyboard either.

7.1 Updating the Wizard Interface

So I wrote the following code. First we have the declaration of the wizard as follows:

```
unit DebuggingToolsWizard;

interface

Uses
  ToolsAPI, Classes;

Type
  TDebuggingWizard = Class(TNotifierObject, IOTAWizard)
  Private
  Protected
    { IOTAWizard }
    Function GetIDString: string;
    Function GetName: string;
    Function GetState: TWizardState;
    Procedure Execute;
    { IOTAMenuWizard }
    {$HINTS OFF}
    Function GetMenuText: string;
    {$HINTS ON}
  Public
    Constructor Create;
    Destructor Destroy; Override;
  End;

  TKeyboardBinding = Class(TNotifierObject, IOTAKeyboardBinding)
  Private
    Procedure AddBreakpoint(const Context: IOTAKeyContext;
      KeyCode: TShortcut; var BindingResult: TKeyBindingResult);
  Protected
    function GetBindingType: TBindingType;
    function GetDisplayName: string;
    function GetName: string;
    procedure BindKeyboard(const BindingServices: IOTAKeyBindingServices);
  Protected
  Public
  End;

  Procedure Register;
  Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
    RegisterProc : TWizardRegisterProc;
    var Terminate: TWizardTerminateProc) : Boolean; StdCall;

Exports
  InitWizard Name WizardEntryPoint;

implementation
```

Here we create a wizard that does nothing (but is needed by the `RegisterProc()` method for DLLs and a keyboard binding class to handle the keyboard input.

Next we need to register both classes with the system so we need to update the `InitialiseWizard()` function as follows:

```
Var
  iWizardIndex : Integer = 0;
  iKeyBindingIndex : Integer = 0;

Function InitialiseWizard(BIDES : IBorlandIDEServices) : TDebuggingWizard;

Begin
  Result := TDebuggingWizard.Create;
  Application.Handle := (BIDES As IOTAServices).GetParentHandle;
  iKeyBindingIndex := (BorlandIDEServices As
       IOTAKeyboardServices).AddKeyboardBinding(
    TKeyboardBinding.Create)
End;
```

... and we need to update the `Initialization` / `Finalization` section as follows:

```
Initialization
Finalization
  If iKeyBindingIndex > 0 Then
    (BorlandIDEServices As
       IOTAKeyboardServices).RemoveKeyboardBinding(iKeyBindingIndex);
  If iWizardIndex > 0 Then
    (BorlandIDEServices As IOTAWizardServices).RemoveWizard(iWizardIndex);
```

7.2 Implementing the Interfaces

So next we'll implement the property getters for the `IOTAKeyboardBinding` interface as follows:

```
function TKeyboardBinding.GetBindingType: TBindingType;
begin
  Result := btPartial;
end;
```

The above indicates it's a partial binding, i.e. adds to the main binding and not an entirely new keyboard binding set like IDE Classic.

```
function TKeyboardBinding.GetDisplayName: string;
begin
  Result := 'Debugging Tools Bindings';
end;
```

This above is the display name that appears under the Key Mappings, Enhancement Modules section in the IDEs option dialogue.

```
function TKeyboardBinding.GetName: string;
begin
  Result := 'DebuggingToolsBindings';
end;
```

And finally this is the unique name for the keyboard binding.

7.3 Implementing the Keyboard Bindings

Next we'll tackle the keyboard bindings with the following code:

```
procedure TKeyboardBinding.BindKeyboard(
  const BindingServices: IOTAKeyBindingServices);
begin
  BindingServices.AddKeyBinding([TextToShortcut('Ctrl+Shift+F8')], AddBreakpoint,
       Nil);
  BindingServices.AddKeyBinding([TextToShortcut('Ctrl+Alt+F8')], AddBreakpoint,
       Nil);
end;
```

Here `Ctrl+Alt+F8` creates / toggles the breakpoint between enabled and disabled and `Ctrl+Shift+F8` creates / edits the breakpoint's properties.

Finally the code that does the actual work as follows:

```
procedure TKeyboardBinding.AddBreakpoint(const Context: IOTAKeyContext;
  KeyCode: TShortcut; var BindingResult: TKeyBindingResult);

var
  i: Integer;
  DS : IOTADebuggerServices;
  MS : IOTAModuleServices;
  strFileName : String;
  Source : IOTASourceEditor;
  CP: TOTAEditPos;
  BP: IOTABreakpoint;

begin
  MS := BorlandIDEServices As IOTAModuleServices;
  Source := SourceEditor(MS.CurrentModule);
  strFileName := Source.FileName;
  CP := Source.EditViews[0].CursorPos;
  DS := BorlandIDEServices As IOTADebuggerServices;
  BP := Nil;
  For i := 0 To DS.SourceBkptCount - 1 Do
    If (DS.SourceBkpts[i].LineNumber = CP.Line) And
      (AnsiCompareFileName(DS.SourceBkpts[i].FileName, strFileName) = 0) Then
      BP := DS.SourceBkpts[i];;
  If BP <> Nil Then
    Begin
      If KeyCode = TextToShortCut('Ctrl+Shift+F8') Then
        BP.Edit(True)
      Else
        BP.Enabled := Not BP.Enabled;
    End Else
    Begin
      BP := DS.NewSourceBreakpoint(strFileName, CP.Line, Nil);
      If KeyCode = TextToShortCut('Ctrl+Alt+F8') Then
        BP.Enabled := False;
      BP.Edit(True);
    End;
  BindingResult := krHandled;
end;
```

This code first searches for a break point at the current line and creates one IF it doesn't exist and displays the Source Breakpoint Properties dialogue for editing the breakpoint's attributes.

I've not described the `TDebuggingWizard` getter methods as these have been described in earlier chapters.

It's not perfect but I think you get the idea and can work out the bugs for yourselves (hint: look what happens when you press `Ctrl+Alt+F8` without an existing breakpoint).

I hope this proves to be useful.

Note: *There is a fix to this code above in The Fix*.

8. The Fix

This was originally published on 11 Feb 2010 using RAD Studio 2009.

For those interested, here's the fix to the previous chapter's code.

```
procedure TKeyboardBinding.AddBreakpoint(const Context: IOTAKeyContext;
  KeyCode: TShortcut; var BindingResult: TKeyBindingResult);

var
  i: Integer;
  DS : IOTADebuggerServices;
  MS : IOTAModuleServices;
  strFileName : String;
  Source : IOTASourceEditor;
  CP: TOTAEditPos;
  BP: IOTABreakpoint;

begin
  MS := BorlandIDEServices As IOTAModuleServices;
  Source := SourceEditor(MS.CurrentModule);
  strFileName := Source.FileName;
  CP := Source.EditViews[0].CursorPos;
  DS := BorlandIDEServices As IOTADebuggerServices;
  BP := Nil;
  For i := 0 To DS.SourceBkptCount - 1 Do
    If (DS.SourceBkpts[i].LineNumber = CP.Line) And
      (AnsiCompareFileName(DS.SourceBkpts[i].FileName, strFileName) = 0) Then
      BP := DS.SourceBkpts[i];;
  If BP = Nil Then
    BP := DS.NewSourceBreakpoint(strFileName, CP.Line, Nil);
  If KeyCode = TextToShortCut('Ctrl+Shift+F8') Then
    BP.Edit(True)
  Else If KeyCode = TextToShortCut('Ctrl+Alt+F8') Then
    BP.Enabled := Not BP.Enabled;
  BindingResult := krHandled;
end;
```

9. Useful Open Tools Utility Functions

This was originally published on 14 Apr 2011 using RAD Studio 2010.

I thought that before tackling some of the more interesting Open Tools API topics previously mentioned that I would describe some useful utility functions that can make using the OTA easier.

9.1 Messages

First I thought we would look at messages (but not custom ones, they will require their own chapter). These are useful for outputting information from your expert and also for helping debug your application without having to run a debugging session of the IDE from within the IDE.

```
Procedure OutputMessage(strText : String);

Begin
  (BorlandIDEServices As IOTAMessageServices).AddTitleMessage(strText);
End;
```

The above code outputs a Title Message to the message window of the IDE. It's only parameter is the message you want displayed in the message window. The message doesn't allow for any interaction, i.e. click on it and going to a line number. For that we need a Tool Message as below:

```
Procedure OutputMessage(strFileName, strText, strPrefix : String;
  iLine, iCol : Integer);

Begin
  (BorlandIDEServices As IOTAMessageServices).AddToolMessage(strFileName,
    strText, strPrefix, iLine, iCol);
End;
```

This is an overloaded version of the first function, so you will require the reserved word overload in the function interface declaration. This procedure has a number of parameters as follows:

- `strFileName` – This is the name of the file to which the message should be associated, i.e. error message from some code where the file name is the file name of the code file (`D:\Path\MyModule.pas`);
- `strText` – This is the message to be displayed;
- `strPrefix` – This is a prefix text that is displayed in front of the message in the message window to define which tool produced the message;
- `iLine` – This is the line number in the file name above at which the message should be associated;
- `iCol` – This is the column number in the file name above at which the message should be associated.

The supplying of the file name, line and column allow the IDE to go to that file, line and column when you double click the message in the message window.

Next we need to be able to clear the messages from the message window. The OTA defines 3 methods for this. I've wrapped them up into a single method which requires an enumerate and set to define which messages are cleared as follows:

```
Type
  (** This is an enumerate for the types of messages that can be cleared. **)
  TClearMessage = (cmCompiler, cmSearch, cmTool);
  (** This is a set of messages that can be cleared. **)
  TClearMessages = Set of TClearMessage;
```

Thus the method can be called with one or more enumerates in the set to define which messages are cleared from the message windows.

```
Procedure ClearMessages(Msg : TClearMessages);
```

```
Begin
  If cmCompiler In Msg Then
    (BorlandIDEServices As IOTAMessageServices).ClearCompilerMessages;
  If cmSearch In Msg Then
    (BorlandIDEServices As IOTAMessageServices).ClearSearchMessages;
  If cmTool In Msg Then
    (BorlandIDEServices As IOTAMessageServices).ClearToolMessages;
End;
```

When working with messages you may wish at a point in the processing of your information to force the message window to be displayed / brought to the foreground. You can do this with the following function:

```
Procedure ShowHelperMessages;

Begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    ShowMessageView(Nil);
End;
```

This displays the main message window of the IDE. In later versions of the IDEs you can output messages to tabs within the message window. To display those messages the above code needs to be modified as follows:

```
Procedure ShowHelperMessages;

Var
  G : IOTAMessageGroup;

Begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    Begin
      G := GetGroup('My Message');
      ShowMessageView(G);
    End;
End;
```

Obviously you can parameterize this method to allow you to pass the name of the message tab to the method to make it more flexible.

So the question arises as to how we add messages to these tabbed message windows?

First you need to create a message group as follows:

```
Var
  MyGroup : IOTAMessageGroup;

Begin
  ...
  MyGroup := AddMessageGroup('My Message Group');
  ...
End;
```

You can also get an existing Message Group as follows:

```
Var
  MyGroup : IOTAMessageGroup;

Begin
  ...
  MyGroup := GetGroup('My Message Group');
  ...
End;
```

The later IDEs have similar overloaded methods to those described above which take an extra parameter which is the Message Group as follows:

```
    procedure AddTitleMessage(const MessageStr: string; const MessageGroupIntf:
       IOTAMessageGroup); overload;
    procedure AddToolMessage(const FileName, MessageStr, PrefixStr: string;
      LineNumber, ColumnNumber: Integer; Parent: Pointer; out LineRef: Pointer;
      const MessageGroupIntf: IOTAMessageGroup); overload;
```

So for instance the first method I described could be re-coded as follows:

```
Procedure OutputMessage(strText : String; strGroupName : String);

Var
  Group : IOTAMessageGroup

Begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    Begin
      Group := GetGroup(strGroupName);
      AddTitleMessage(strText, Group);
    End;
End;
```

It should be noted that all the above functions do not check to see if the `BorlandIDEServices` interface is available. For most OTA code this shouldn't be a problem as if this services isn't available, you've got bigger problems with the IDE than not being able to use the interface. The only situation I can think of off the top of my head where this could be a problem is if you call this when creating a splash screen for BDS/RAD Studio 2005 and above as this code get called before the `BorlandIDEServices` service is available. But since the message window isn't available you wouldn't be able to log messages.

9.2 Projects and Project Groups

The following code samples provide mean by which you can get access to project groups, projects, modules and editor code. This is not a comprehensive list, but other code will appear in the other chapters that should fill in the missing gaps.

```
Function ProjectGroup: IOTAProjectGroup;

Var
  AModuleServices: IOTAModuleServices;
  AModule: IOTAModule;
  i: integer;
  AProjectGroup: IOTAProjectGroup;

Begin
  Result := Nil;
  AModuleServices := (BorlandIDEServices as IOTAModuleServices);
  For i := 0 To AModuleServices.ModuleCount - 1 Do
    Begin
      AModule := AModuleServices.Modules[i];
      If (AModule.QueryInterface(IOTAProjectGroup, AProjectGroup) = S_OK) Then
       Break;
    End;
  Result := AProjectGroup;
end;
```

The above code returns a reference to the project group (there is only 1 in the IDE at a time). If there is no group open (i.e. nothing in the Project Manager) then this will return nil.

```
Function ActiveProject : IOTAProject;

var
  G : IOTAProjectGroup;

Begin
  Result := Nil;
  G := ProjectGroup;
```

```
  If G <> Nil Then
    Result := G.ActiveProject;
End;
```

This above code returns a reference to the active project in the Project Manager (the project highlighted in bold in the tree view). If there is no active project then this function will return nil.

```
Function ProjectModule(Project : IOTAProject) : IOTAModule;

Var
  AModuleServices: IOTAModuleServices;
  AModule: IOTAModule;
  i: integer;
  AProject: IOTAProject;

Begin
  Result := Nil;
  AModuleServices := (BorlandIDEServices as IOTAModuleServices);
  For i := 0 To AModuleServices.ModuleCount - 1 Do
    Begin
      AModule := AModuleServices.Modules[i];
      If (AModule.QueryInterface(IOTAProject, AProject) = S_OK) And
        (Project = AProject) Then
        Break;
    End;
  Result := AProject;
End;
```

The above code returns a reference to the projects source modules (DPR, DPK, etc) for the given project.

```
Function ActiveSourceEditor : IOTASourceEditor;

Var
  CM : IOTAModule;

Begin
  Result := Nil;
  If BorlandIDEServices = Nil Then
    Exit;
  CM := (BorlandIDEServices as IOTAModuleServices).CurrentModule;
  Result := SourceEditor(CM);
End;
```

The above code returns a reference to the active IDE source editor interface. If there is no active editor then this method returns nil.

```
Function SourceEditor(Module : IOTAModule) : IOTASourceEditor;

Var
  iFileCount : Integer;
  i : Integer;

Begin
  Result := Nil;
  If Module = Nil Then Exit;
  With Module Do
    Begin
      iFileCount := GetModuleFileCount;
      For i := 0 To iFileCount - 1 Do
        If GetModuleFileEditor(i).QueryInterface(IOTASourceEditor,
          Result) = S_OK Then
          Break;
    End;
End;
```

The above code provides a reference to the given modules source editor interface.

```
Function EditorAsString(SourceEditor : IOTASourceEditor) : String;

Var
  Reader : IOTAEditReader;
  iRead : Integer;
  iPosition : Integer;
  strBuffer : AnsiString;

Begin
  Result := '';
  Reader := SourceEditor.CreateReader;
  Try
    iPosition := 0;
    Repeat
      SetLength(strBuffer, iBufferSize);
      iRead := Reader.GetText(iPosition, PAnsiChar(strBuffer), iBufferSize);
      SetLength(strBuffer, iRead);
      Result := Result + String(strBuffer);
      Inc(iPosition, iRead);
    Until iRead < iBufferSize;
  Finally
    Reader := Nil;
  End;
End;
```

Lastly, the above code returns the given editor source's code as a string.

All of the above are not meant to be a complete set of utilities, just those that I've coded for my *Browse and Doc It* and *Integrated Testing Helper* IDE experts. It does make me think now that I should combine these into a single utility module across all my IDE experts (there currently in different project specific modules with only a couple of duplications).

The code for this can be found attached to this PDF as `OTABrowseAndDocIt.zip` and `OTAIntegratedTestingHelper.zip`.

10. Open Tools API Custom messages

This was originally published on 01 May 2011 using RAD Studio 2010.

In my last chapter on the OTA I showed some methods for handling messages in the Open Tools API. What I didn't cover in the post was custom messages.

I use custom message in my *Integrated Testing Helper* IDE plug-in to colour message depending upon the success or failure of the command-line tools being run. This way the user gets immediate feedback when looking at the messages as to what has failed.

There are four interfaces for custom messages. These are `IOTACustomMessage`, `IOTACustomMessage50`, `IOTACustomMessage100` and `IOTACustomDrawMessage`. The first three are to do with the management of message information and the fourth (and the more interesting one for me) deals with custom drawing the message.

The first interface `IOTACustomMessage` defines properties and method for handling line numbers, column numbers, file name and source text for message that require associations with code in the IDE. The second interface `IOTACustomMessage50` defines properties and methods for dealing with nested messages (will return to this before the end of this chapter). The third interface `IOTACustomMessage100` provides methods to allow a message to override the default IDE mechanism for navigating to a file and allowing the message to do this instead. The fourth interface `IOTACustomDrawMessage` allow the message to draw itself allowing for a change of font name, style, size and colour (you can also change the background).

While it's in my mind, there is one thing to ensure your expert does when using custom message and that is clear those custom messages from the message windows and tabs before being unloaded. If you do not do this and say unload a BPL, your message view will cause access violation because it can't find the drawing code for the messages.

First we'll look at the definition of a custom message that's used in my *Integrated Testing Helper* IDE plug-in as below:

```
TDGHCustomMessage = Class(TInterfacedObject, IOTACustomMessage,
    INTACustomDrawMessage)
Private
  FMsg : String;
  FFontName   : String;
  FForeColour : TColor;
  FStyle : TFontStyles;
  FBackColour : TColor;
  FMessagePntr : Pointer;
Protected
  Procedure SetForeColour(iColour : TColor);
  // INTACustomDrawMessage Methods
  function CalcRect(Canvas: TCanvas; MaxWidth: Integer; Wrap: Boolean): TRect;
  procedure Draw(Canvas: TCanvas; const Rect: TRect; Wrap: Boolean);
  // IOTACustomMessage Methods
  function GetColumnNumber: Integer;
  function GetFileName: string;
  function GetLineNumber: Integer;
  function GetLineText: string;
  procedure ShowHelp;
Public
  Constructor Create(strMsg : String; FontName : String;
    ForeColour : TColor = clBlack; Style : TFontStyles = [];
    BackColour : TColor = clWindow);
  Property ForeColour : TColor Write SetForeColour;
  Property MessagePntr : Pointer Read FMessagePntr Write FMessagePntr;
End;
```

Here we define a class that implements 2 of the 4 interfaces. Since I'm not interested in child message handling in this way (because it doesn't help me with earlier versions of Delphi that don't have this

functionality) or changing the default message handling then I'll concentrate on the first and the last interfaces.

The class we defined must implement all the methods of the interfaces referenced in the inheritance list, thus we have to implement methods for returning line, column, filename and source text for the first interface and methods to handling drawing for the second interface (fourth above). Additionally I've defined 2 more properties to allow me to change the colour of the message and manage the messages pointer reference (this reference is used for nesting the messages which we will tackle later). The reason for not using the `TCustomMessage50` interface for handling child messages is that this tool needs to work with earlier version of Delphi so I actual manage this information myself.

10.1 IOTACustomMessage Methods

Below are the implemented methods for the first interface:

```
function TDGHCustomMessage.GetColumnNumber: Integer;
begin
  Result := 0;
end;

function TDGHCustomMessage.GetFileName: string;
begin
  Result := '';
end;

function TDGHCustomMessage.GetLineNumber: Integer;
begin
  Result := 0;
end;

function TDGHCustomMessage.GetLineText: string;
begin
  Result := FMsg;
end;

procedure TDGHCustomMessage.ShowHelp;
begin
  //
end;
```

Since I don't want to browse source code with these messages they simply return default information so that double clicking them has no effect.

Next, let's look at the constructor:

```
constructor TDGHCustomMessage.Create(strMsg: String; FontName : String;
  ForeColour : TColor = clBlack; Style : TFontStyles = [];
  BackColour : TColor = clWindow);

Const
  {$IFNDEF D2009}
  strValidChars : Set Of Char = [#10, #13, #32..#128];
  {$ELSE}
  strValidChars : Set Of AnsiChar = [#10, #13, #32..#128];
  {$ENDIF}

Var
  i : Integer;
  iLength : Integer;

begin
  SetLength(FMsg, Length(strMsg));
  iLength := 0;
  For i := 1 To Length(strMsg) Do
    {$IFNDEF D2009}
    If strMsg[i] In strValidChars Then
    {$ELSE}
```

```
    If CharInSet(strMsg[i], strValidChars) Then
    {$ENDIF}
      Begin
        FMsg[iLength + 1] := strMsg[i];
        Inc(iLength);
      End;
  SetLength(FMsg, iLength);
  FFontName := FontName;
  FForeColour := ForeColour;
  FStyle := Style;
  FBackColour := BackColour;
  FMessagePntr := Nil;
end;
```

The constructor has a conditional define in it. This is simply to make the code work with different versions of Delphi as Delphi 2009's strings were changed to UniCode. The constructor simply assigns the passed message to an internal string first removing any carriage returns or line feeds and extended characters (as they display as boxes) and initialises the custom message.

10.2 INTACustomDrawMessage Methods

Now we can move onto the drawing code. The drawing code is in 2 parts, first the calculation of the size of the message to be drawn and then the drawing itself. We'll first look at the `CalcRect()` method as below:

```
function TDGHCustomMessage.CalcRect(Canvas: TCanvas; MaxWidth: Integer;
  Wrap: Boolean): TRect;

begin
  Canvas.Font.Name := FFontName;
  Canvas.Font.Style := FStyle;
  Result:= Canvas.ClipRect;
  Result.Bottom := Result.Top + Canvas.TextHeight('Wp');
  Result.Right := Result.Left + Canvas.TextWidth(FMsg);
end;
```

First the font name and font style are assigned to the messages canvas reference, then canvas rectangle is obtained and adjusted to suit the height and width of the message with the given font and style.

Next the actual drawing code as follows:

```
procedure TDGHCustomMessage.Draw(Canvas: TCanvas; const Rect: TRect;
  Wrap: Boolean);

begin
  If Canvas.Brush.Color = clWindow Then
    Begin
      Canvas.Font.Color := FForeColour;
      Canvas.Brush.Color := FBackColour;
      Canvas.FillRect(Rect);
    End;
  Canvas.Font.Name := FFontName;
  Canvas.Font.Style := FStyle;
  Canvas.TextOut(Rect.Left, Rect.Top, FMsg);
end;
```

First the colours are assigned and the background rectangle rendered then the canvas is set with the message's font name and style and finally the message is drawn on the canvas. Note that I do not use the `wrap` parameter. You could wrap you messages to fit in the window width but to do this you would then have to change the `CalcRect()` method and the `Draw()` method to first calculate the height of the wrapped message with the Win32 API `DrawText()` method and then draw the message with the same API call.

10.3 Creating the messages

Now we have a custom message we need a way to add these messages to the message window. I do this with the following procedure:

```
Function AddMsg(strText : String; boolGroup, boolAutoScroll : Boolean;
  FontName : String; ForeColour : TColor; Style : TFontStyles;
  BackColour : TColor = clWindow; Parent : Pointer = Nil) : Pointer;

Var
  M : TDGHCustomMessage;
  G : IOTAMessageGroup;

begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    Begin
      M := TDGHCustomMessage.Create(strText, FontName, ForeColour, Style,
       BackColour);
      Result := M;
      If Parent = Nil Then
        Begin
          G := Nil;
          If boolGroup Then
            G := AddMessageGroup(strITHelperGroup)
          Else
            G := GetMessageGroup(0);
          {$IFDEF D2005}
          If boolAutoScroll <> G.AutoScroll Then
            G.AutoScroll := boolAutoScroll;
          M.MessagePntr := AddCustomMessagePtr(M As IOTACustomMessage, G);
          {$ELSE}
          AddCustomMessage(M As IOTACustomMessage, G);
          {$ENDIF}
        End Else
          {$IFDEF D2005}
          AddCustomMessage(M As IOTACustomMessage, Parent);
          {$ELSE}
          AddCustomMessage(M As IOTACustomMessage);
          {$ENDIF}
    End;
end;
```

Firstly, this method is design to work in multiple version of Delphi, hence the conditional compilation. Delphi 2005 and above support nested messages but earlier version do not. Additionally, the IDE handles nested messages in a way that's not what you would expect. The methods to add a message return a pointer to the message NOT a reference to the custom message class and it's this pointer you need to nest messages.

The method creates the custom message. It will also assigns the message to the *ITHelper*'s message group (remember this is code from *ITHelper*). Finally it adds the message to the IDE using the appropriate message method. Some of the message adding methods take the parent pointer and some do not. I assign the messages pointer value to the message's MessagePntr property so that it can be referenced later IF I want to nest more messages.

So there it is.

The code for this can be found attached to this PDF as `OTAIntegratedTestingHelper.zip`.

11. Open Tools API Interface Version Guide for Backward Compatability

This was originally published on 21 Jul 2011 using RAD Studio 2010.

There are a number of reasons why I write these blogs. The first is simply to write down what I've found out so that I can reference it later (years generally) but also to impart what I have found to the rest of the community as they have generally helped me over the years to find the solutions I have needed. Finally, for the Open Tools API (OTA), there is little documentation.

Consequently, I was looking at code folding and unfolding in the IDE yesterday and thought what versions of Delphi (and C++ Builder) does this support (I'll be writing another blog about this soon). A little digging around and consolidating a number of sources produced the following list of interface number conventions.

```
Delphi 1 = IOTAXxxx10
Delphi 2 = IOTAXxxx20
Delphi 3 = IOTAXxxx30
Delphi 4 = IOTAXxxx40
Delphi 5 = IOTAXxxx50
Delphi 6 = IOTAXxxx60
Delphi 7 = IOTAXxxx70
Delphi 8 = IOTAXxxx80
Delphi 2005 = IOTAXxxx90
Delphi 2006 = IOTAXxxx100
Delphi 2007 = IOTAXxxx110
Delphi 2009 = IOTAXxxx120
Delphi 2010 = IOTAXxxx140
Delphi XE = IOTAXxxx150
Delphi XE2 = IOTAXxxx160
Delphi XE3 = IOTAXxxx170
Delphi XE4 = IOTAXxxx180
Delphi XE5 = IOTAXxxx190
Delphi XE6 = IOTAXxxx200
Delphi XE7 = IOTAXxxx210
```

I think if I remember far enough back, the Open Tools API started with Delphi 3 (OTAXxxx30) therefore you will never find interfaces for Delphi 1, 2 or 3.

Why do you need to know this? Well, if you are doing things for just yourself then probably you don't need to know, however if you are going to release you OTA expert/wizard to the world at large, creating something that only works in Delphi 2010 for instance, is of no use to those who have earlier versions. To code up an OTA expert/wizard for multiple version of Delphi you will probably find that you need to use condition compilation `{$IFDEF VER120}` etc. That's another blog I need to write about soon :-)

UPDATE @ 27 Jul 2011

I've been looking at the `ToolsAPI` files across different version of Delphi and have come to the conclusion that the above does not always hold true. For instance the `IOTAElideActions120` interface is defined in Delphi 2010 but exists as `IOTAElideActions` in Delphi 2006, 2007 and 2009.

Conclusion, the best way to test your code against different compilers is simply to compile it using that version :-)

12. Conditional Compilation of Open Tools API experts

This was originally published on 21 Jul 2011 using RAD Studio 2010.

Ordinarily in conditional formatting you would write something like...

```
Procedure MyProc;

Begin
  DoSomething;
  {$IFDEF VER120}
  DoSomethingElse;
  {$ENDIF}
  DoSomethingAgain;
End;
```

However, if you want this piece of code to work with a particular version of Delphi and later version of Delphi then this will not work as these types of definitions are specific to a version of Delphi. You could use the conditional `IF` statement, but this is not supported by earlier version of Delphi.

My solution to this was the below include file.

```
(**

  An include unit to define compiler definitions for version of delphi. If the
  definition exists it means that that version of the compiler or high is
  available.

  @Version 1.0
  @Date    26 May 2010
  @Author  David Hoyle

**)
{$IFDEF VER90}
  {$DEFINE D0002}
{$ENDIF}

{$IFDEF VER93}
  {$DEFINE D0002}
{$ENDIF}

{$IFDEF VER100}
  {$DEFINE D0002}
  {$DEFINE D0003}
{$ENDIF}

{$IFDEF VER110}
  {$DEFINE D0002}
  {$DEFINE D0003}
{$ENDIF}

{$IFDEF VER120}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
{$ENDIF}

{$IFDEF VER125}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
{$ENDIF}

{$IFDEF VER130}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
```

```
{$ENDIF}

{$IFDEF VER140}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
{$ENDIF}

{$IFDEF VER150}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
{$ENDIF}

{$IFDEF VER160}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
{$ENDIF}

{$IFDEF VER170}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
{$ENDIF}

{$IFDEF VER180}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
{$ENDIF}

{$IFDEF VER190}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
{$ENDIF}

{$IFDEF VER200}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
```

```
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
{$ENDIF}

{$IFDEF VER210}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
{$ENDIF}
```

What this allows you to do is use conditional compilation against a Delphi version number and it also means that that piece of code will work with that version and all higher versions. For the above snippet of code would change to…

```
Procedure MyProc;

Begin
  DoSomething;
  {$IFDEF D0004}
  DoSomethingElse;
  {$ENDIF}
  DoSomethingAgain;
End;
```

… signifying that this will work in Delphi 4 and above.

To include this file (towards the top of your code) use the `{$INCLUDE CompilerDefintions.inc}` statement.

For examples of how this can be used in the OTA have a look at the *Integrated Testing Helper* Wizard code.

The code for this can be found attached to this PDF as `OTAIntegratedTestingHelper.zip`.

13. Handling Folding and Unfolding code in the IDE

This was originally published on 21 Jul 2011 using RAD Studio 2010.

First of all I need to apologise for a mistake in the original article on setting up an expert/wizard where by a package based wizard would not load correctly in the IDE. What is missing from the code is the exporting of the Register procedure in the interface section of the wizard module as shown below. Once this is added the wizard menu will appear under the Help menu of the IDE.

```
Unit IDEFoldedBitsWizard;

Interface

Uses
  ToolsAPI;

Type
  TIDEFoldedBitsWizard = Class(TInterfacedObject, IOTAWizard, IOTAMenuWizard)
  public
    ...
  End;

  Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
    RegisterProc : TWizardRegisterProc;
    var Terminate: TWizardTerminateProc) : Boolean; StdCall;

  Procedure Register; // MISSING LINE

Exports
  InitWizard Name WizardEntryPoint;

Implementation
...
```

Now for code folding.

Handling code folding is a little more difficult that other OTA things as the `BorlandIDEServices` does not export an interface from any of its sub-interfaces. By this I mean you cannot get an `IOTAElideActions` interface from any property or method directly. So to get the interface we have to ask another interface whether it supports it. Luckily for us the clue is in the `ToolsAPI.pas` file where it says this interface is present in `IOTAEditView` interface.

What we need to do (as shown below) is ask the `IOTAEditView` interface if it has the `IOTAElideActions` interface and give us a reference to it.

```
procedure TIDEFoldedBitsWizard.Execute;

var
  TopView: IOTAEditView;
  I : IOTAElideActions;

begin
  TopView := (BorlandIDEServices As IOTAEditorServices).TopView;
  If TopView.QueryInterface(IOTAElideActions, I) = S_OK Then
    Begin
      I.ToggleElisions;
      TopView.Paint;
    End;
end;
```

What this piece of code does it toggle the folding and unfolding of the code at the current cursor position in the active editor. We use the `QueryInterface` method to obtain the new interface (so long as it returns `S_OK`) and then we can use it as any other interface. One thing I've found with folding / unfolding code in the editor is that it DOES NOT update the interface until you move something, therefore you need to force this by calling the `IOTAEditView's` Paint method.

13.1 IOTAElideActions120 Methods

Now for a quick explanation of the different methods available.

The `IOTAElideActions120` interface implements the following (available from Delphi 2006 onwards):

- `ElideNearestBlock`: This folds the block in which the cursor currently sits;
- `UnElideNearestBlock`: This unfolds the block in which the cursor currently sits;
- `UnElideAllBlocks`: This folds ALL blocks in the editor;
- `EnableElisions`: This enables code folding.

The `IOTAElideActions` interface implements the following (available from Delphi 2010 onwards):

- `ToggleElisions`: This folds and unfolds the current block in which the cursor resides;
- `ElideNamespaces`: This folds all Namespaces in the currently active module;
- `ElideRegions`: This folds all Regions in the currently active module;
- `ElideTypes`: This folds all Types in the currently active module;
- `ElideNestedProcs`: This folds all Nested Procedures and Functions in the currently active module;
- `ElideGlobals`: This folds all Globally defined variables in the currently active module;
- `ElideMethods`: This folds all Methods in the currently active module;

Unfortunately there is no way to test whether you are in a nested block, however if you call the `IOTAEditView's` Paint method when moving and displaying a new cursor position this should automatically open any folded code and draw the screen correctly.

14. IDE Compilation Events

This was originally published on 05 Aug 2011 using RAD Studio 2010.

The interface(s) we are going to look at are the `IOTAIDENotifierXxx` interfaces. Now when implementing these interfaces it's important to implement ALL the inherited interfaces, else the methods signatures of these versions will NOT be called. I've recently made this mistake again and wondered for an hour why my Delphi 2010 project manager menu didn't appear. It was simply because I didn't place the `IOTALocalMenu` interface in the class heritage list :-(The above is especially important if you want the expert to work in earlier version of Delphi.

I'm using Delphi 2010 at the moment, so the interfaces available to me in `ToolAPI.pas` are as follows:

- `IOTAIDENotifier` (I think this has always been available)
- `IOTAIDENotifier50` (Available from Delphi 5 onwards)
- `IOTAIDENotifier80` (Available from Delphi 2005 onwards)

So we now need to define our wizard interface with these interfaces as follows:

```
  TTestingHelperWizard = class(TNotifierObject, IOTANotifier, IOTAIDENotifier,
    IOTAIDENotifier50 {$IFDEF D2005}, IOTAIDENotifier80 {$ENDIF}, IOTAWizard)
  {$IFDEF D2005} Strict {$ENDIF} Private
  {$IFDEF D2005} Strict {$ENDIF} Protected
  Public
    Constructor Create;
    Destructor Destroy; Override;
    // IOTAWizard methods
    procedure Execute;
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    // IOTAIDENotifier methods
    Procedure FileNotification(NotifyCode : TOTAFileNotification;
      Const FileName : String; var cancel : Boolean);
    procedure BeforeCompile(const Project: IOTAProject; var Cancel: Boolean);
       overload;
    procedure AfterCompile(Succeeded: Boolean); overload;
    // IOTAIDENotifier50 methods
    procedure BeforeCompile(const Project: IOTAProject; IsCodeInsight: Boolean;
      var Cancel: Boolean); Overload;
    procedure AfterCompile(Succeeded: Boolean; IsCodeInsight: Boolean); overload;
    // IOTAIDENotifier80 methods
    {$IFDEF D2005}
    procedure AfterCompile(const Project: IOTAProject; Succeeded:
      Boolean; IsCodeInsight: Boolean); Overload;
    {$ENDIF}
  Published
  end;
```

You will notice that the `IOTANotifier` interface is not specifically implemented. This is handled automatically by implementing the class with the TNotifierObject parent class which handles all those method of `IOTANotifier`.

You will note that the newer versions of Delphi provide different overloaded version of the same methods for `BeforeCompile` and `AfterCompile`. These newer versions expose additional information that these IDEs provide, for instance, later IDEs trigger these method for CodeInsight, therefore if you only want to handle these methods for proper compilation then you need to ignore the calls `IF CodeInsight` is true.

I'm not going to go into the whole implementation of the wizard, you can refer to the earlier chapter Starting an Open Tools API Project.

14.1 IOTAIDENotifierXxx methods

Now for an explanation of the interface methods (I will not go through the `IOTAWizard` interface methods here as they are already covered in the above article):

- `BeforeCompile`: These methods are called before each project is compiled. Place in this method any checks or processing you want to do before the project is compiled;
- `AfterCompile`: These methods are called after each compilation. Place in this method any checks or processing you want to do after the project is compiled
- `FileNotification`: This method is called for various file operations with the IDE. The `NotifyCode` parameter is an enumerate that tells you what type of operation has occurred as follows:
  - `ofnFileOpening`: the file passed in `FileName` is opening where `FileName` is the name of the file being opened;
  - `ofnFileOpened`: the file passed in `FileName` has opened where `FileName` is the name of the file that was opened;
  - `ofnFileClosing`: the file passed in `FileName` is closing where `FileName` is the name of the file being closed;
  - `ofnDefaultDesktopLoad`: I haven't found when this is triggered in my test but I assume it is when the IDE loads the Default Desktop settings;
  - `ofnDefaultDesktopSave`: I haven't found when this is triggered in my test but I assume it is when the IDE saves the Default Desktop settings;
  - `ofnProjectDesktopLoad`: this is triggered when the IDE loads a project's desktop settings where `FileName` is the name of the desktop settings file;
  - `ofnProjectDesktopSave`: this is triggered when the IDE saves a project's desktop settings where `FileName` is the name of the desktop settings file;
  - `ofnPackageInstalled`: this is triggered when a package (BPL) list loaded by the IDE where `FileName` is the name of the BPL file;
  - `ofnPackageUninstalled`: this is triggered when a package (BPL) list unloaded by the IDE where `FileName` is the name of the BPL file;
  - `ofnActiveProjectChanged`: this is triggered when a project is made active in the IDE's Project Manager where the `FileName` is the project file (`.dproj`, `.bdsproj` or `.dpr` for older version of Delphi);

You can stop the IDE from doing anything with these files by changing the Cancel parameter of the method to `TRUE`.

For a real life implementation of these interfaces (from which these code was extracted) please download and look at my *Integrated Testing Helper* IDE expert.

The code for this can be found attached to this PDF as `OTAIntegratedTestingHelper.zip`.

15. Debugging Open Tools API Experts on a Windows 7 64-bit Machine

This was originally published on 05 Aug 2011 using RAD Studio 2010.

Since upgrading to a new machine (Dell Precision M6500 with Windows 7 64-bit) late last year I've been unable to debug Open Tools API experts in a second instance of the IDE. When I've tried to run the second IDE it starts up okay and then the first IDE breaks like it's at a breakpoint. I press F9 to continue and it happens a number of other times until the loading of the second IDE throws a complete wobbly and crashes taking out the first IDE along with it. I wasn't impressed and wondered if I'd made a big mistake in getting a 64-bit machine.

Oh, and by the way, we hadn't got anywhere near loading my expert so that wasn't the problem as the second IDE was a complete new instance started with the command line `BDS.exe -pDelphi -rBrowseAndDocItDPR`. The first just loads the Delphi personality and the second creates a complete new registry area for this second IDE so it doesn't interfere with the first.

Anyway I decided this evening to try running the IDE in a compatibility mode (Windows XP Service Pack 2). It shouldn't need it as Delphi 2010 (and RAD Studio) where supposed to be built for Windows 7. Well it made a difference. The loading of the second IDE still caused the first to break several times BUT pressing F9 allowed it to load completely (with several other F9s) and I can now debug the experts.

Hope this helps anyone else how comes across this problem.

Dave.

PS: Hasn't solved my other debugging problem with my Eidolon DLL in Excel and Excel not finding a printer or the debugger attaching to the DLL.

16. IDE Compilation Events – Revisited...

This was originally published on 10 Aug 2011 using RAD Studio 2010.

As a continuation of the previous post on compiler events (IDE Compilation Events), in Delphi 2010 onwards there are 2 new interfaces named `IOTACompileServices` and `IOTACompileNotifier` and I thought I would describe how to use it and what it does.

We'll deal with the services interface first as we'll need this to understand this to implement the other interface. The `IOTACompileServices` interface is exposed by the `BorlandIDEServices` global variable, so we can get a reference to this compile services by simply casting this variable as we've done for various other interfaces as shown below:

```
Function InitialiseWizard : TTestingHelperWizard;

Begin
  ...
  {$IFDEF D2010}
  iCompiler := (BorlandIDEServices As IOTACompileServices).AddNotifier(
    TCompilerNotifier.Create);
  {$ENDIF}
  ...
End;
```

This interface exposes the following methods:

- `AddNotifier`: This adds a `IOTACompilerNotifier` to the IDE so that it can handle compile events;
- `CancelBackgroundCompile`: This method cancels a background compilation. It waits for background thread to terminate before returning and can optionally prompt the user as to whether the background compilation should be terminated and returns `True` if the thread was cancelled and False if not;
- `CompileProjects`: This method compiles a list of projects defined by an array of `IOTAProject` interfaces supplied. The method returns `crOTABackground` if background compiling is enabled. You need to implement a `IOTACompileNotifier` to be informed of the result of background compilation;
- `DisableBackgroundCompilation`: This method prevent any subsequent compilation requests to the IDE from taking place in the background thread, regardless of settings IDE settings;
- `EnableBackgroundCompilation`: This method re-enables the compilation of projects in the background thread;
- `IsBackgroundCompileActive`: This method returns `True` if background compilation is active else returns False;
- `RemoveNotifier`: This method removes a `IOTACompileNotifier` object from the IDE using the index number returned by `AddNotifer`.

Now for the notifier. First we need to define a class which implements the notifier interface `IOTACompileNotifier` and then get the IDE to use it as follows:

```
  TCompilerNotifier = Class(TNotifierObject, IOTACompileNotifier)
  {$IFDEF D2005} Strict {$ENDIF} Private
  {$IFDEF D2005} Strict {$ENDIF} Protected
    Procedure ProjectCompileStarted(const Project: IOTAProject; Mode:
      TOTACompileMode);
    Procedure ProjectCompileFinished(const Project: IOTAProject; Result:
      TOTACompileResult);
    Procedure ProjectGroupCompileStarted(Mode: TOTACompileMode);
    Procedure ProjectGroupCompileFinished(Result: TOTACompileResult);
  Public
  End;
  {$ENDIF}
```

Below are some simple implementations that output messages through a custom function in *ITHelper* called `DebugMsg`. You could substitute another message handler as described in previous posts.

```
Const
  strCompileMode : Array[Low(TOTACompileMode)..High(TOTACompileMode)] Of String = (
    'Make', 'Build', 'Check', 'Make Unit');
  strCompileResult : Array[Low(TOTACompileResult)..High(TOTACompileResult)] of
      String = (
    'Failed', 'Succeeded', 'Background');

Procedure TCompilerNotifier.ProjectCompileStarted(const Project: IOTAProject; Mode:
      TOTACompileMode);

Begin
  DebugMsg(Format('Compile Started (%s)...', [strCompileMode[Mode]]), Project);
End;

Procedure TCompilerNotifier.ProjectCompileFinished(const Project: IOTAProject;
      Result: TOTACompileResult);

Begin
  DebugMsg(Format('Compile Finished (%s)...', [strCompileResult[Result]]), Project);
End;

Procedure TCompilerNotifier.ProjectGroupCompileStarted(Mode: TOTACompileMode);

Begin
  DebugMsg(Format('Group Compile Finished (%s)...', [strCompileMode[Mode]]));
End;

Procedure TCompilerNotifier.ProjectGroupCompileFinished(Result: TOTACompileResult);

Begin
  DebugMsg(Format('Group Compile Finished (%s)...', [strCompileResult[Result]]));
End;
```

The first bit of code in the article shows how to add the notifier to the IDE but we must remove it from the IDE on close down as follows:

```
  ...
Initialization
Finalization
  (BorlandIDEServices As IOTACompileServices).RemoveNotifier(iCompiler);
End;
```

So the question now is how do these events behave?

The order of these events are straight forward and I'll describe them for a project that has multiple projects where there are dependencies and thus multiple projects compiled at a time:

- `ProjectGroupCompileStarted` is called first where the `Mode` parameter returning `Make` for a straight forward compile, `Build` for a Build and `Check` for a syntax check. I'm not sure how in the Delphi 2010 you can make a single unit, but I presume that `MakeUnit` would be returned for this;
- `ProjectCompileStarted` is called before each project is compiled with the Mode parameter returning the same values as above;
- `ProjectCompileFinished` is called after each project is compiled with the Result parameter returning whether the compilation was successful, failed or is in the background;
- `ProjectGroupCompileFinished` is called after all the projects are compiled (regardless of whether they have succeeded or failed) and returns the overall result of the compilation, thus if a project has failed then this will return a Failure. Consequently if all projects are compiled successfully then this will return Success.

I hope this helps :-)

17. Finding Open Tools API Interfaces

This was originally published on 11 Aug 2011 using RAD Studio 2010.

I've seen recently while looking for OTA information many people asking how to get a particular interface from the IDE. What follows is a process that I follow to try and find these interfaces and could be used as a rule of thumb but is not always correct.

17.1 Services Interfaces

Most of the `IOTAXxxxxServices` interfaces are exposed through casting the `BorlandIDEServices` global variable as follows:

```
Procedure Something;

Var
  CompileServices : IOTACompileServices;

Begin
  CompileServices := (BorlandIDEServices As IOTACompileServices);
  CompileServices.DisableBackgroundCompilation;
End;
```

There is one known exception to this and that is the `IOTASplashScreenServices` interface. This is exposed by another global variable `SplashScreenServices`. The reason for this other variable is that the `BorlandIDEServices` variable at the point in time when the splash screen is being displayed is not set up and available.

17.2 Finding Interfaces

This method I use to find interfaces is quite simple. For example, let's take the `IOTAEditView` interface. I'll explain in a minute why I was looking for this interface as it's an exception to the rule of thumb I'm describing here. Anyway, what I need to do is find another interfaces that has a property or method that returns this interface. You can do this via a number of methods. You can use a Find/Search method in the IDE to searching the `ToolsAPI.pas` file. My preferred method is to use a key-stroke exposed by GExperts (`Ctrl+Alt+Up` or `Ctrl+Alt+Down`) on the interface name and these keystrokes will move you to the previous or next instance of this interface in the source code.

The `IOTAEditView` interface is exposed by the following interfaces and methods / properties:

- `IOTASourceEditor70.GetEditView` a getter method for the property `IOTASourceEditor70.EditViews`;
- `IOTAEditBuffer60.GetTopView` a getter method for the property `IOTAEditBuffer60.TopView`;
- `IOTAEditorServices60.GetTopView` a getter method for the property `IOTAEditorServices60.TopView`.

The above give us 3 paths to this interface. The last one is completely resolved such that we can get the interface with the following code:

```
Procedure Something;

Var
  EditView : IOTAEditView;
  CP : TOTAEditPos;

Begin
  EditView := (BorlandIDEServices As IOTAEditorServices);
  CP := EditView.CursorPos;
  OutputDebugString(PChar(Format('Line %d, Column %d', [CP.Line, CP.Col])));
End;
```

For the other 2 in the list above we would need to repeat the exercise of finding the interface by looking for interfaces and their method that return the secondary interface. So for instance with the first item in the list above we would have to look for methods and property that returns a `IOTASourceEditor` interface.

The reason that I don't look for the interface with the number on the end is that in each release of Delphi the interfaces that are implemented by `BorlandIDEServices` implements the highest interface without the number which in turn implements these previous IDE versions of the interfaces.

The above could be resolved in the following manner:

```
Procedure Something;

Var
  CM : IOTAModule;
  i : Integer;
  SourceEditor : IOTASourceEditor;

Begin
  CM := (BorlandIDEServices as IOTAModuleServices).CurrentModule;
  For i := 0 To CM.ModuleFileCount - 1 Do
    If ModuleFileEditors[i].QueryInterface(IOTASourceEditor, SourceEditor) = S_OK
       Then
      Begin
        EditView := SourceEditor.EditViews[0];
        CP := EditView.CursorPos;
        OutputDebugString(PChar(Format('Line %d, Column %d', [CP.Line, CP.Col])));
        Break;
      End;
End;
```

This one is a bit awkward as the `IOTAModule` interface property `ModuleFileEditors` only returns a `IOTAEditor` interface BUT is actually a `IOTASourceEditor` interface. To get the interface we must query the `IOTAEditor` interface to see if it does implement `IOTASourceEditor` which we can then use.

Now I come back to the reason for wanting the `IOTAEditView` interface is that this interface according to the comments in `ToolsAPI.pas` exposes the `IOTAElideServices` interface for folding and unfolding code but like the above example the interface is not exposed explicitly and must be obtained through a `QueryInterface` call as below:

```
EV := (BorlandIDEServices As IOTAEditorServices).TopView;
{$IFDEF D2006}
EV.QueryInterface(IOTAElideActions, EA);
{$ENDIF}
C.Col := iIdentCol;
C.Line := iIdentLine;
{$IFDEF D2006}
If EA <> Nil Then
  EA.UnElideNearestBlock;
{$ENDIF}
```

I hope this helps people gain better access to the Open Tools API.

18. Editor Notifiers

This was originally published on 19 Aug 2011 using RAD Studio 2010.

This time around I'm going to look at editor notifiers so that you can get notifications of when editors have changed and a few other useful things.

The implementation below I'm going to show will only work in Delphi 2005 and above. I will write something at the end on how you can work around this for earlier version of Delphi in the same manner that I've done for my *Browse and Doc It* expert.

This implementation is a little different from the previous ones I've described as it's a native IDE interface signified by the N in the name rather than the O, however we declare the notifier class in exactly the same way as before as shown below:

```
  TEditorNotifier = Class(TNotifierObject, INTAEditServicesNotifier)
  Strict Private
  Strict Protected
  Public
    procedure WindowShow(const EditWindow: INTAEditWindow; Show, LoadedFromDesktop:
      Boolean);
    procedure WindowNotification(const EditWindow: INTAEditWindow; Operation:
      TOperation);
    procedure WindowActivated(const EditWindow: INTAEditWindow);
    procedure WindowCommand(const EditWindow: INTAEditWindow; Command, Param:
      Integer; var Handled: Boolean);
    procedure EditorViewActivated(const EditWindow: INTAEditWindow; const EditView:
      IOTAEditView);
    procedure EditorViewModified(const EditWindow: INTAEditWindow; const EditView:
      IOTAEditView);
    procedure DockFormVisibleChanged(const EditWindow: INTAEditWindow; DockForm:
      TDockableForm);
    procedure DockFormUpdated(const EditWindow: INTAEditWindow; DockForm:
      TDockableForm);
    procedure DockFormRefresh(const EditWindow: INTAEditWindow; DockForm:
      TDockableForm);
    Constructor Create;
    Destructor Destroy; Override;
  End;
```

Now that we've defined the class we need to tell the IDE how to use the notifier. If you haven't already looked at the chapter Starting an Open Tools API Project in this series you may wish to, to understand the following code.

The below code registers the notifier with the IDE returning an integer value which is needed to remove the notifier from the IDE at the end of the session.

```
Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices; RegisterProc :
      TWizardRegisterProc; var Terminate: TWizardTerminateProc) : Boolean; StdCall;

Begin
  Application.Handle := Application.MainForm.Handle;
  Result := BorlandIDEServices <> Nil;
  If Result Then
    Begin
      ...
      iEditorIndex := (BorlandIDEServices As IOTAEditorServices).AddNotifier(
        TEditorNotifier.Create);
      ...
    End;
End;
```

Finally the below code removes the notifier from the IDE with the previously obtained integer value:

```
  ...
Initialization
```

```
  ...
Finalization
  ...
  (BorlandIDEServices As IOTAEditorServices).RemoveNotifier(iEditorIndex);
  ...
End.
```

There are a lot of methods to this notifier that are fired by the IDE for different situations. I'm going to list each method and list below it the situation in which the IDE fires the method and what the parameters that are provide contain.

18.1 EditorViewActivated

```
procedure TEditorNotifier.EditorViewActivated(const EditWindow: INTAEditWindow;
      const EditView: IOTAEditView);
```

This method is fired each time a tab is changed in the editor whether that's through opening and closing files or simply changing tabs to view a different file. The `EditWindow` parameter provides access to the editor window. This is usually the first docked editor window unless you've opened a new editor window to have a second one visible. The `EditView` parameter provides you with access to the view of the file where you can get information about the cursor positions, the selected block, etc. By drilling down through the `Buffer` property you can get the text associated with the file but that's for another chapter, then next one I think.

18.2 EditorViewModified

```
procedure TEditorNotifier.EditorViewModified(const EditWindow: INTAEditWindow; const
      EditView: IOTAEditView);
```

This method is fired each time the text of the file is changed whether that is an insertion or a deletion of text. The values returned by the parameters are the same as those for the above `EditorViewActivated` method.

18.3 WindowActivated

```
procedure TEditorNotifier.WindowActivated(const EditWindow: INTAEditWindow);
```

Well I've been unable to get this to fire in both a docked layout and a classic undocked layout, so if someone else knows what fires this, please let me know.

18.4 WindowCommand

```
procedure TEditorNotifier.WindowCommand(const EditWindow: INTAEditWindow; Command,
      Param: Integer; var Handled: Boolean);
```

This method is fired for some keyboard commands but there doesn't seem to be any logic to when it is fired or for what. The `Command` parameter is the command number and in all my tests the `Param` parameter was 0. I've check against keyboard bindings and have found that this event is not fired for OTA keyboard binding.

18.5 WindowNotification

```
procedure TEditorNotifier.WindowNotification(const EditWindow: INTAEditWindow;
      Operation: TOperation);
```

This event is fired for each editor window that is opened or closed. The `EditWindow` parameter is a reference to the specific editor window opening or closing and the `Operation` parameter depicts whether the editor is opening (insert) or closing (remove).

18.6 WindowShow

```
procedure TEditorNotifier.WindowShow(const EditWindow: INTAEditWindow; Show,
      LoadedFromDesktop: Boolean);
```

This event is fired each time an editor window appears or disappear. The `EditWindow` parameter references the editor changing appearance with the `Show` parameter defining whether it is appearing (`show = true`) or disppearing (`show = false`). The `LoadFromDesktop` parameter defines whether the operation is being caused by a desktop layout being loaded or not.

18.7 DockFormRefresh

```
procedure TEditorNotifier.DockFormRefresh(const EditWindow: INTAEditWindow;
      DockForm: TDockableForm);
```

This method seems to be fired when the IDE is closing down and the desktop of being save. I've not been able to get the event to fire for any other situations. The `EditWindow` is the edit window that the docking operation is be docked to (it's a dock site) and `DockForm` is the form that is being docked.

18.8 DockFormUpdated

```
procedure TEditorNotifier.DockFormUpdated(const EditWindow: INTAEditWindow;
      DockForm: TDockableForm);
```

This event seems to be fired when a dockable form is docked with an Edit Window dock site. The parameters are the same as those for the above `DockFormRefresh`.

18.9 DockFormVisibleChanged

```
procedure TEditorNotifier.DockFormVisibleChanged(const EditWindow: INTAEditWindow;
      DockForm: TDockableForm);
```

This method seems to be fired when desktops are loaded and not as I thought when dockable forms change their visibility. The parameters are the same as those for the above `DockFormRefresh`.

With just the `EditorViewActivated` and `EditorViewModified` we can understand what editor files are being shown and display information based on that. We can also know when a file has been updated (changed) so that the information can be updated. This is how the *Browse and Doc It* expert works and can display the explorer view of the code in the files.

For earlier IDEs we have to do something else. What we have to do is set up a timer and look for changes to the active file in the editor (see `ActiveProject` and `ActiveSourceEditor` in a previous chapter: Useful Open Tools Utility Functions) and react to those changes when the project or file change. To detect the modification of the file itself, then we need to monitor the size of the edit buffer each time we check and look for changes. You can see all of this in the source for to the *Browse and Doc It* expert.

The code for this can be found attached to this PDF as `OTABrowseAndDocIt.zip`.

19. Aboutbox Plugins and Splash Screens

This was originally published on 25 Aug 2011 using RAD Studio 2010.

This time around I'm going to talk about the ability of Delphi 2005 (and above) to display information about your wizard/expert on the splash screen and also in the About dialogue box.

First of all we need some global private module variables to hold information that is common between the about box plugin and the splash screen. The reason for them outside a method is they need to be available to the Initialization section of the module. These variables are as follows:

```
Implementation

...

Var
  {$IFDEF D2005}
  VersionInfo       : TVersionInfo;
  bmSplashScreen    : HBITMAP;
  iAboutPluginIndex : Integer = 0;
  {$ENDIF}

...
```

In the above variable declaration there is a user-defined type `TVersionInfo` which has the following definition:

```
  Type
    TVersionInfo = Record
      iMajor  : Integer;
      iMinor  : Integer;
      iBugfix : Integer;
      iBuild  : Integer;
    End;
```

This reason for this is not that it is a requirement of the Open Tools API but simply because I want to display the version number of the expert in the about dialogue and on the splash screen.

Next we need some constants and resource strings for displaying the information on the splash screen and in the about dialogue as follows:

```
{$IFDEF D2005}
Const
  strRevision : String = ' abcdefghijklmnopqrstuvwxyz';

ResourceString
  strSplashScreenName = 'My Expert Title %d.%d%s for Embarcadero RAD Studio ####';
  strSplashScreenBuild = 'Freeware by Author (Build %d.%d.%d.%d)';
{$ENDIF}
```

Now we can look at the creation of About box information in the Wizard Initialization function. Firstly, the code loads a bitmap from the experts (DLL/BPL) resource file for display in the about box and on the splash screen. This bitmap should be 24 x 24 in size with the lower left hand side depicting the transparent colour. I've found that a white background works best if you want a single image to work on all the different splash screens. You can create this bitmap and add it to the RES file directly using the Borland Image Editor (or similar) or use a resource file to create the RES file which should then be included.

The resource file looks like this:

```
SplashScreenBitMap BITMAP "Images\SplashScreenIcon.bmp"
```

and should be named with the .RC extension. This can then be included in the main wizard / expert module as follows:

```
{$R '..\SplashScreenIcon.res' '..\SplashScreenIcon.RC'}
```

This bitmap is then passed to the `AddPluginInfo` method to create the about box plugin entry. You don't have to be as complicated as I've made it below but the below shows you how to display the experts version number. Note that this method expects that the `VersionInfo` record is already populated with version information. This happens in the `Initialization` section of the module along with the splash screen.

```
Function InitialiseWizard : TWizardTemplate;

Var
  Svcs : IOTAServices;

Begin
  Svcs := BorlandIDEServices As IOTAServices;
  ToolsAPI.BorlandIDEServices := BorlandIDEServices;
  Application.Handle := Svcs.GetParentHandle;
  {$IFDEF D2005}
  // Aboutbox plugin
  bmSplashScreen := LoadBitmap(hInstance, 'SplashScreenBitMap');
  With VersionInfo Do
    iAboutPluginIndex := (BorlandIDEServices As IOTAAboutBoxServices).AddPluginInfo(
      Format(strSplashScreenName, [iMajor, iMinor, Copy(strRevision, iBugFix + 1,
       1)]),
      '$WIZARDDESCRIPTION$.',
      bmSplashScreen,
      False,
      Format(strSplashScreenBuild, [iMajor, iMinor, iBugfix, iBuild]),
      Format('SKU Build %d.%d.%d.%d', [iMajor, iMinor, iBugfix, iBuild]));
  {$ENDIF}
  ...
End;
```

Next we will look at the splash screen information. This is somewhat different to the rest of the Open Tools API because the `BorlandIDEServices` variable at the point in time the modules are loaded is not necessarily set and available. To facilitate this, Borland / Codegear added a specific service interface for the splash screen called `SplashScreenServices`. The below code is in the Initialization section of the module so that it occurs as the DLL/BPL is being loaded and not when (later) the wizard is initialised. The code below gets the build information for the expert, then gets a reference to the resource image as above and passes all this information to the `AddPluginBitmap` method of the `IOTASplashScreenServices`.

```
...
Initialization
  {$IFDEF D2005}
  BuildNumber(VersionInfo);
  // Add Splash Screen
  bmSplashScreen := LoadBitmap(hInstance, 'SplashScreenBitMap');
  With VersionInfo Do
    (SplashScreenServices As IOTASplashScreenServices).AddPluginBitmap(
      Format(strSplashScreenName, [iMajor, iMinor, Copy(strRevision, iBugFix + 1,
       1)]),
      bmSplashScreen,
      False,
      Format(strSplashScreenBuild, [iMajor, iMinor, iBugfix, iBuild]));
  {$ENDIF}
Finalization
  {$IFDEF D2005}
  // Remove Aboutbox Plugin Interface
  If iAboutPluginIndex > 0 Then
    (BorlandIDEServices As
       IOTAAboutBoxServices).RemovePluginInfo(iAboutPluginIndex);
  {$ENDIF}
End.
```

Obviously the missing piece is how to get the version information from the DLL/BPL. Below is the code to do this:

```
Procedure BuildNumber(Var VersionInfo: TVersionInfo);

Var
  VerInfoSize: DWORD;
  VerInfo: Pointer;
  VerValueSize: DWORD;
  VerValue: PVSFixedFileInfo;
  Dummy: DWORD;
  strBuffer: Array [0 .. MAX_PATH] Of Char;

Begin
  GetModuleFileName(hInstance, strBuffer, MAX_PATH);
  VerInfoSize := GetFileVersionInfoSize(strBuffer, Dummy);
  If VerInfoSize <> 0 Then
    Begin
      GetMem(VerInfo, VerInfoSize);
      Try
        GetFileVersionInfo(strBuffer, 0, VerInfoSize, VerInfo);
        VerQueryValue(VerInfo, '\', Pointer(VerValue), VerValueSize);
        With VerValue^ Do
          Begin
            VersionInfo.iMajor := dwFileVersionMS Shr 16;
            VersionInfo.iMinor := dwFileVersionMS And $FFFF;
            VersionInfo.iBugfix := dwFileVersionLS Shr 16;
            VersionInfo.iBuild := dwFileVersionLS And $FFFF;
          End;
      Finally
        FreeMem(VerInfo, VerInfoSize);
      End;
    End;
End;
```

I hope that was all straight forward. From this point onwards I will be putting all the code that I've written about on the Open Tools API into a single project and can be compiled by the following versions of Delphi:

- Delphi 5
- Delphi 7
- Delphi 2006
- Delphi 2009
- Delphi 2010

as these are the only versions I have, however I'm trying to ensure that the code will be correctly conditionally compiled to work with all other version of Delphi above Delphi 5. This code can be used as the basis of your own expert and has been built modularly so that the discrete topics are in the main in their own module so you can pick and choose the bits you want.

The code for this can be found attached to this PDF as `OTAChapter19AboutboxPluginsAndSplashScreens.zip`.

20. Reading editor code

This was originally published on 29 Aug 2011 using RAD Studio 2010.

In this chapter I'll go through how to get the code from the editor and do something with it. For this example I'm simply going to allow you to select a method from the current module and move to its position. This will use some of the utility functions I've described before in (see the chapter: Useful Open Tools Utility Functions) but I'll describe how they work.

All the code for this is attached in a zip file at the end of this article.

To invoke this code I'm going to extend the key bindings we introduced in the chapter: Key Bindings and Debugging Tools.

Below I've added a new key binding attached to a new handler.

```
Procedure TKeybindingTemplate.BindKeyboard(Const BindingServices:
      IOTAKeyBindingServices);

Begin
  ...
  BindingServices.AddKeyBinding([TextToShortcut('Ctrl+Shift+Alt+F9')],
      SelectMethodExecute, Nil);
End;
```

The handler is implemented to call a method to select a method as below.

```
Procedure TKeybindingTemplate.SelectMethodExecute(Const Context: IOTAKeyContext;
  KeyCode: TShortcut; Var BindingResult: TKeyBindingResult);

Begin
  SelectMethod;
  BindingResult := krHandled;
End;
```

Below is the implementation of the `SelectMethod` method.

```
Procedure SelectMethod;

Var
  slItems: TStringList;
  SE: IOTASourceEditor;
  CP: TOTAEditPos;
  recItemPos : TItemPosition;
  iIndex: Integer;

Begin
  slItems := TStringList.Create;
  Try
    GetMethods(slItems);
    iIndex := TfrmItemSelectionForm.Execute(slItems, 'Select Method');
    If iIndex > -1 Then
      Begin
        SE := ActiveSourceEditor;
        If SE <> Nil Then
          Begin
            recItemPos.Data := slItems.Objects[iIndex];
            CP.Line := recItemPos.Line;
            CP.Col := 1;
            SE.EditViews[0].CursorPos := CP;
            SE.EditViews[0].Center(CP.Line, 1);
          End;
      End;
  Finally
    slItems.Free;
  End;
End;
```

There's a lot in here so I'll try and break it down into stages. This method creates a string list in which we will add the methods to be selected from. This string list is passed to the method `GetMethods` to extract the methods from the source code. We will look at this in more detail later. This string list is then passed to a form which I'm not going to describe here but is included in the code at the bottom of the article. All you need to know is the forms method takes the string list and returns the index of the selected it in the string list. The rest of the above method moves the cursor position to the line number of the selected method (the line number is stored in the `Object` member of the string list.) I'll explain how this is done below but you can look up the technique in Storing numbers, enumerates and sets in a TStringList all at the same time.

```
Type
  TSubItem = (siData, siPosition);

  TItemPosition = Record
    Case TSubItem Of
      siData: (Data : TObject);
      siPosition: (
        Line   : SmallInt;
        Column : SmallInt
      );
  End;
```

The idea about the above record is that you define a variant record where by the `Line` and `Column` information, decsribed as `SmallInts` (16 bits each) are described in the same memory location as the 32 bit `TObject` pointer. This way the `Line` and `Column` information is stored in the lower and upper portions of the `Data` reference. This also means we do not need casting and shifting commands as we just assign the `TStringList`'s `TObject` data to the record's `Data` memory and then we can access the `Line` and `Column` information directly from the record and visa versa.

The first method called in the `SelectMethod` method above is `GetMethods` as shown below:

```
Procedure GetMethods(slItems : TStringList);

Var
  SE: IOTASourceEditor;
  slSource: TStringList;
  i: Integer;
  recPos : TItemPosition;
  boolImplementation : Boolean;
  iLine: Integer;

Begin
  SE := ActiveSourceEditor;
  If SE <> Nil Then
    Begin
      slSource := TStringList.Create;
      Try
        slSource.Text := EditorAsString(SE);
        boolImplementation := False;
        iLine := 1;
        For i := 0 To slSource.Count - 1 Do
          Begin
            If Not boolImplementation Then
              Begin
                If Pos('implementation', LowerCase(slSource[i])) > 0 Then
                  boolImplementation := True;
              End Else
            If IsMethod(slSource[i]) Then
              Begin
                recPos.Line := iLine;
                recPos.Column := 1;
                slItems.AddObject(slSource[i], recPos.Data);
              End;
            Inc(iLine);
          End;
        slItems.Sort;
```

```
      Finally
        slSource.Free;
      End;
    End;
End;
```

The first thing this method does is get a reference to the current Source Editor in the IDE using the utility function `ActiveSourceEditor` below:

```
Function ActiveSourceEditor : IOTASourceEditor;

Var
  CM : IOTAModule;

Begin
  Result := Nil;
  If BorlandIDEServices = Nil Then
    Exit;
  CM := (BorlandIDEServices as IOTAModuleServices).CurrentModule;
  Result := SourceEditor(CM);
End;
```

This method uses the `IOTAModuleServices` interface to get the IDE's current module being edited. It then uses another utility function `SourceEditor` to return the `IOTASourceEditor` interface from this current module provided as below:

```
Function SourceEditor(Module : IOTAModule) : IOTASourceEditor;

Var
  iFileCount : Integer;
  i : Integer;

Begin
  Result := Nil;
  If Module = Nil Then Exit;
  With Module Do
    Begin
      iFileCount := GetModuleFileCount;
      For i := 0 To iFileCount - 1 Do
        If GetModuleFileEditor(i).QueryInterface(IOTASourceEditor,
          Result) = S_OK Then
          Break;
    End;
End;
```

The above code cycles through the module files associated with the given module looking for a `IOTASourceEditor` interface. When found this interface is returned. This function uses the `QueryInterface` method to test for the interface we want. There are a number of occasion in the Open Tools API where the required interface is not directly exposed and the use of `QueryInterface` is required to test for the interface we require.

Returning to the `GetMethods` method, if we have a valid source editor interface we then need the source code associated with that editor using the `SourceEditor` function as follows:

```
Function EditorAsString(SourceEditor : IOTASourceEditor) : String;

Const
  iBufferSize : Integer = 1024;

Var
  Reader : IOTAEditReader;
  iRead : Integer;
  iPosition : Integer;
  strBuffer : AnsiString;

Begin
```

```
  Result := '';
  Reader := SourceEditor.CreateReader;
  Try
    iPosition := 0;
    Repeat
      SetLength(strBuffer, iBufferSize);
      iRead := Reader.GetText(iPosition, PAnsiChar(strBuffer), iBufferSize);
      SetLength(strBuffer, iRead);
      Result := Result + String(strBuffer);
      Inc(iPosition, iRead);
    Until iRead < iBufferSize;
  Finally
    Reader := Nil;
  End;
End;
```

This function gets a `Reader` interface from the source editor by calling the `CreateReader` method. To get the text from the `Reader` interface we need to get the text in chunks using the `GetText` method of the `IOTAEditReader` interface. The buffer must be a `AnsiChar` buffer as the editor only returns Unicode UTF8 code or ANSI code not double bit Unicode. We loop the `GetText` method until it returns a number (the number of characters read) less than the buffer size and we add the buffer contents to the end of the resultant string for the function. We must maintain throughout this the position in the text (`iPosition`) as the `GetText` method returns the chunk of text starting at a position.

Returning again now to `GetMethods`, we can assign the source editor text to the `Text` property of a string list and cycle through these strings searching for method headings in the implementation section. To help with that I created the below function to check a line of text for method heading.

```
Function IsMethod(strLine : String) : Boolean;

Const
  strMethods : Array[1..4] Of String = ('procedure', 'function', 'constuctor',
       'destructor');

Var
  i : Integer;

Begin
  Result := False;
  For i := Low(strMethods) To High(strMethods) Do
    If Pos(strMethods[i], LowerCase(strLine)) > 0 Then
      Begin
        Result := True;
        Break;
      End;
End;
```

If we find a line that contains a method heading, then this line is added to the string list passed to `GetMethods`.

Back to `SelectMethod`, we then pass the string list to our form code for selection by the user. Once the user has selected the method they want we get the line number of that method from the string lists `Objects` property and using the active source editors `EditViews` property move the cursor position and centre the editor on that line.

The code for this can be found attached to this PDF as `OTAChapter20ReadingEditorCode.zip`.

21. Writing editor code

This was originally published on 01 Sep 2011 using RAD Studio 2010.

In this chapter I'm going to extend the ideas from Chapter 10: Reading editor code and make the code insert a comment block above the method declaration and move the cursor to the start of the description in the comment.

To do this I've modified the `SelectMethod` as below:

```
Procedure SelectMethod;

Var
  slItems: TStringList;
  SE: IOTASourceEditor;
  CP: TOTAEditPos;
  iIndex: Integer;

Begin
  slItems := TStringList.Create;
  Try
    GetMethods(slItems);
    iIndex := TfrmItemSelectionForm.Execute(slItems, 'Select Method');
    If iIndex > -1 Then
      Begin
        CP := InsertComment(slItems, iIndex);
        SE := ActiveSourceEditor;
        If SE <> Nil Then
          Begin
            SE.EditViews[0].CursorPos := CP;
            SE.EditViews[0].Center(CP.Line, CP.Col);
          End;
      End;
  Finally
    slItems.Free;
  End;
End;
```

Note that the `InsertComment` method (detailed in a minute) returns a new cursor position which we then update. The code that got the cursor position from our user-defined record has been moved into the start of the `InsertComment` method as below:

```
Function InsertComment(slItems : TStringList; iIndex : Integer) : TOTAEditPos;

Var
  recItemPos : TItemPosition;
  SE: IOTASourceEditor;
  Writer: IOTAEditWriter;
  i: Integer;
  iIndent: Integer;
  iPosition: Integer;
  CharPos : TOTACharPos;

Begin
  recItemPos.Data := slItems.Objects[iIndex];
  Result.Line := recItemPos.Line;
  Result.Col := 1;
  SE := ActiveSourceEditor;
  If SE <> Nil Then
    Begin
      Writer := SE.CreateUndoableWriter;
      Try
        iIndent := 0;
        For i := 1 To Length(slItems[iIndex]) Do
          If slItems[iIndex][i] = #32 Then
            Inc(iIndent)
          Else
```

```
            Break;
          CharPos.Line := Result.Line;
          CharPos.CharIndex := Result.Col - 1;
          iPosition := SE.EditViews[0].CharPosToPos(CharPos);
          Writer.CopyTo(iPosition);
          OutputText(Writer, iIndent, '(**'#13#10);
          OutputText(Writer, iIndent, #13#10);
          OutputText(Writer, iIndent, '  Description.'#13#10);
          OutputText(Writer, iIndent, #13#10);
          OutputText(Writer, iIndent, '  @precon  '#13#10);
          OutputText(Writer, iIndent, '  @postcon '#13#10);
          OutputText(Writer, iIndent, #13#10);
          OutputText(Writer, iIndent, '**)'#13#10);
          Inc(Result.Line, 2);
          Inc(Result.Col, iIndent + 2);
        Finally
          Writer := Nil;
        End;
      End;
End;
```

In this method the following happens:

- Update the Result of the function with the current method cursor position;
- Next get a reference to the active source editor as described in the previous chapter;
- If the source editor is valid obtain an undoable writer interface. The IDE supports both an undoable interface and one without that can't be undone. I prefer to allow the user to undo any changes so always use the `CreateUndoableWriter` method.
- Next we count the number of space at the start of the method line to see if we need to indent the comment (tabs are not supported in this example);
- Then we need to convert the cursor position to a position index into the writer buffer using the `EditViews.CharPosToPos` method. One thing to note is that the `CharPosToPos` function uses a `TOTACharPos` record not a `TOTAEditPos` record, so we need to convert. In converting the char index in the `TOTACharPos` record is 0 based NOT 1 based therefore we need to subtract 1;
- Once we have the position in the writer buffer of the cursor position we use the `CopyTo` method of the writer to move to that position;
- We then output the lines of the comment using a new utility function `OutputText` which is described below;
- Finally we update the Result cursor position for the new position in the inserted comment text.

The reason for refactoring the `OutputText` functionality out is that this is different for Delphi 2009 (Unicode) and above so casting is required to stop compiler warnings appearing and it also makes the code cleaner.

```
Procedure OutputText(Writer : IOTAEditWriter; iIndent : Integer; strText : String);

Begin
  {$IFNDEF D2009}
  Writer.Insert(PAnsiChar(StringOfChar(#32, iIndent) + strText));
  {$ELSE}
  Writer.Insert(PAnsiChar(AnsiString(StringOfChar(#32, iIndent) + strText)));
  {$ENDIF}
End;
```

There is a caveat to working with the `IOTAEditWriter` interface in that it is a move forward only buffer – you cannot move backwards in the buffer. This causes a problem with multiple inserts as generally to maintain the cursor position you would want to move backwards through the text so you don't have to update all you cursor positions. The way I get round this is to perform the insert at each location (going backwards) with new instances of the `IOTAEditWriter` interface.

22. Fatal Mistake in DLL... Doh!

This was originally published on 06 Sep 2011 using RAD Studio 2010.

I've made a mistake in the code that I've been attaching to the last few OTA chapters. I was positive at the time that it all worked but I've found an issue that needs to be resolve and will appear in the next chapter.

The issue is the difference between how a DLL wizard and a Package wizard MUST be initialised. My mistake is that for a DLL I've allowed the main wizard to be registered twice and the IDE complains and stops loading and initialising the experts. The main wizard can only be added to the system in a package. It's better to show you than try and explain it as below:

```
Function InitialiseWizard(WizardType : TWizardType) : TWizardTemplate;

Var
  Svcs : IOTAServices;

Begin
  ...
  If WizardType = wtPackageWizard Then // Only register main wizard this way if
       PACKAGE
    iWizardIndex := (BorlandIDEServices As IOTAWizardServices).AddWizard(Result);
  ...
End;
```

The above is ONLY called for a Package as the DLL is register with the IDE differently as will be shown below.

I've defined a new enumerate to help with this as follows:

```
Type
  TWizardType = (wtPackageWizard, wtDLLWizard);
```

This therefore means a number of changes in the way the Package and the DLLs are called as follows:

```
procedure Register;

begin
  InitialiseWizard(wtPackageWizard);
end;

Function InitWizard(Const BorlandIDEServices : IBorlandIDEServices;
  RegisterProc : TWizardRegisterProc;
  var Terminate: TWizardTerminateProc) : Boolean; StdCall;

Begin
  Result := BorlandIDEServices <> Nil;
  If Result Then
    RegisterProc(InitialiseWizard(wtDLLWizard));
End;
```

As you can see the DLL `InitWizard` procedure calls the `RegisterProc` internal IDE procedure. I've tried not to call this and it fails so you MUST register your DLLs in this way.

All I can say is Doh! Should have checked better.

23. Project Repository Wizards

This was originally published on 07 Sep 2011 using RAD Studio 2010.

In this chapter I'm going to describe how to implement a Project Wizard which appears in the File | New or File | New | Other dialogue depending on the version of Delphi you have. I've actually had to work this one out from scratch as the last time I did this was with Delphi 3 which used a different method for creating wizards.

This particular wizard will either appear as an icon in a tab of its own for earlier version of Delphi or under a new branch in modern versions of Delphi. These types of wizard are useful for creating new projects in their entirety.

Below is the code to create the wizard interface and it contains quite a number of interfaces. The first `IOTAWizard` should be obvious from previous example but this is then followed by various `IOTARepositoryWizard` and `IOTAProjectWizard` interfaces. The different versions of the interfaces are for different version of the IDE and their methods will be explained later. To implement a Project Wizard you must also implement the Repository Wizard interfaces as well.

```
Type
  TRepositoryWizardInterface = Class(TNotifierObject, IOTAWizard,
      IOTARepositoryWizard
    {$IFDEF D0006}, IOTARepositoryWizard60 {$ENDIF}
    {$IFDEF D2005}, IOTARepositoryWizard80 {$ENDIF},
    IOTAProjectWizard
    {$IFDEF D2005}, IOTAProjectWizard100 {$ENDIF})
  {$IFDEF D2005} Strict {$ENDIF} Private
  {$IFDEF D2005} Strict {$ENDIF} Protected
  Public
    {$IFDEF D2005}
    Constructor Create;
    {$ENDIF}
    // IOTAWizard
    Procedure Execute;
    Function GetIDString: String;
    Function GetName: String;
    Function GetState: TWizardState;
    Procedure AfterSave;
    Procedure BeforeSave;
    Procedure Destroyed;
    Procedure Modified;
    // IOTARepositoryWizard
    Function GetAuthor: String;
    Function GetComment: String;
    {$IFDEF D0006}
    Function GetGlyph: Cardinal;
    {$ELSE}
    Function GetGlyph: HICON;
    {$ENDIF}
    Function GetPage: String;
    {$IFDEF D0006}
    // IOTARepositoryWizard60
    Function GetDesigner: String;
    {$ENDIF}
    {$IFDEF D2005}
    // IOTARepositoryWizard80
    Function GetGalleryCategory: IOTAGalleryCategory;
    Function GetPersonality: String;
    {$ENDIF}
    // IOTAProjectWizard
    {$IFDEF D2005}
    // IOTAProjectWizard100
    Function IsVisible(Project: IOTAProject): Boolean;
    {$ENDIF}
  End;
```

There is one thing to note here. The `GetGlyph` declaration has changed since Delphi 5.

Below I'm going to walk through each method of the above definition so you can understand what needs coding and what doesn't and how they should be implemented.

Below are some resource strings. These are not need for the OTA but simply for me to implement some messages in the code to help workout when methods are called.

```
{$IFDEF D0006}
ResourceString
  strRepositoryWizardGroup = 'Repository Wizard Messages';
{$ENDIF}
{$IFDEF D2005}
ResourceString
  strMyCustomCategory = 'OTA Custom Gallery Category';
{$ENDIF}
```

This is a method of the `IOTAWizard` interface and as far as I can tell does not get called for this type of wizard.

```
Procedure TRepositoryWizardInterface.AfterSave;

Begin
  OutputMessage('AfterSave' {$IFDEF D0006}, strRepositoryWizardGroup {$ENDIF});
End;
```

This is a method of the IOTAWizard interface and as far as I can tell does not get called for this type of wizard.

```
Procedure TRepositoryWizardInterface.BeforeSave;

Begin
  OutputMessage('BeforeSave' {$IFDEF D0006}, strRepositoryWizardGroup {$ENDIF});
End;
```

This create method is only implemented for Delphi 2005 and above as the IDE works differently in earlier versions. This constructor creates a new Category in the gallery under which the project Wizard is installed. For Delphi 2005 and above, this is the method that should be used not the below `GetPage` method from older version of the IDE. It simply adds a new category to the IDE with the Delphi New Category as its parent.

```
{$IFDEF D2005}
Constructor TRepositoryWizardInterface.Create;

Begin
  With (BorlandIDEServices As IOTAGalleryCategoryManager) Do
    Begin
      AddCategory(FindCategory(sCategoryDelphiNew), strMyCustomCategory,
        'OTA Custom Gallery Category');
    End;
End;
{$ENDIF}
```

This is a method of the `IOTAWizard` interface and as far as I can tell does not get called for this type of wizard.

```
Procedure TRepositoryWizardInterface.Destroyed;

Begin
  OutputMessage('Destroyed' {$IFDEF D0006}, strRepositoryWizardGroup {$ENDIF});
End;
```

This method is executed when the Project Wizard is selected from the Gallery and this is where we will in future chapters implements the creation of a project.

```
Procedure TRepositoryWizardInterface.Execute;

Begin
  ShowMessage('Hello OTA Example from the Project Repository Wizard.');
End;
```

This is a method of the IOTAWizard interface and returns the Author of the wizard.

```
Function TRepositoryWizardInterface.GetAuthor: String;

Begin
  Result := 'David Hoyle';
End;
```

This is a method of the IOTAWizard interface and returns a comment for the wizard.

```
Function TRepositoryWizardInterface.GetComment: String;

Begin
  Result := 'This is an example of an OTA Repository Wizard';
End;
```

This is a method of the IOTARepositoryWizard60 interface and returns the type of designer to be used. This is due to the IDEs of the time being able to target Linux. For this we just return the constant string for the VCL. This will perhaps change in Delphi XE2 and the targeting of the Mac OS.

```
{$IFDEF D0006}
Function TRepositoryWizardInterface.GetDesigner: String;

Begin
  Result := dVCL;
End;
{$ENDIF}
```

This is a method of the IOTARepositoryWizard80 interface and specifies under which category in the gallery this new Project Wizard will appear. In this case it appears under the new category we created in the constructor of our wizard.

```
{$IFDEF D2005}
Function TRepositoryWizardInterface.GetGalleryCategory: IOTAGalleryCategory;

Begin
  Result := (BorlandIDEServices As
      IOTAGalleryCategoryManager).FindCategory(strMyCustomCategory);
End;
{$ENDIF}
```

This is a method of the IOTARepositoryWizard interface and defines the ICON handle to be used for the Project Wizard. In testing I've ascertained that this can ONLY be an ICON and not a bitmap. I should have guessed from the original signature. All we do here is return the handle of an ICON in a resource bound to the Package or DLL. You can see how this is done by looking at the source code at the end of this article.

```
{$IFDEF D0006}
Function TRepositoryWizardInterface.GetGlyph: Cardinal;
{$ELSE}
Function TRepositoryWizardInterface.GetGlyph: HICON;
{$ENDIF}

Begin
  Result := LoadIcon(hInstance, 'RepositoryWizardProjectIcon')
End;
```

This is a method of the IOTAWizard interface and returns the ID string of the wizard. This must be unique especially in a project that contains multiple wizards.

```
Function TRepositoryWizardInterface.GetIDString: String;

Begin
  Result := 'OTA.Repository.Wizard.Example';
End;
```

This is a method of the `IOTAWizard` interface and returns the name of the wizard.

```
Function TRepositoryWizardInterface.GetName: String;

Begin
  Result := 'OTA Repository Wizard Example';
End;
```

This is a method of the `IOTARepositoryWizard` interface and is required for earlier version of Delphi (before 2005) in order to tell the IDE on which page (tab) the Project Wizard should appear.

```
Function TRepositoryWizardInterface.GetPage: String;

Begin
  Result := 'OTA Examples';
End;
```

This is a method of the `IOTARepositoryWizard80` interface and tells the IDE which personality the Project belongs to (Delphi, C++ Builder, etc).

```
{$IFDEF D2005}
Function TRepositoryWizardInterface.GetPersonality: String;

Begin
  Result := sDelphiPersonality;
End;
{$ENDIF}
```

This is a method of the `IOTAWizard` interface and returns that the wizard is enabled.

```
Function TRepositoryWizardInterface.GetState: TWizardState;

Begin
  Result := [wsEnabled];
End;
```

This is a method of the `IOTAProjectWizard100` interface which signifies that the wizard is visible for all projects. You may wish to disable a project wizard for a particular given project.

```
{$IFDEF D2005}
Function TRepositoryWizardInterface.IsVisible(Project: IOTAProject): Boolean;

Begin
  Result := True;
End;
{$ENDIF}
```

This is a method of the `IOTAWizard` interface and as far as I can tell does not get called for this type of wizard.

```
Procedure TRepositoryWizardInterface.Modified;

Begin
  OutputMessage('Modified' {$IFDEF D0006}, strRepositoryWizardGroup {$ENDIF});
End;
```

Finally we need to remove any message from the IDE before we unload. This is only because I'm output messages with the library routines. If you don't use message that implement `IOTACustomMessages` you do not require this but I always think it's a safe thing to do.

```
Initialization
Finalization
  ClearMessages([cmCompiler..cmTool]);
End.
```

Obviously we need to create our wizard so the following code is added to the `InitialiseWizard` function. Note that this isn't the main wizard for this example project and therefore is not passed back to the DLL loading code. If it were the main wizard then you would only use `AddWizard` in a Package and `RegisterProc` called from `InitWizard` in a DLL. See the post Fatal Mistake in DLL... Doh! for more details

```
Function InitialiseWizard(WizardType : TWizardType) : TWizardTemplate;

Var
  Svcs : IOTAServices;

Begin
  ...
  // Create Project Repository Interface
  iRepositoryWizardIndex := (BorlandIDEServices As IOTAWizardServices).AddWizard(
    TRepositoryWizardInterface.Create);
End;
```

And of course we need to unload the wizard on unloading the package.

```
  ...
Initialization
  ...
Finalization
  ...
  // Remove Repository Wizard Interface
  If iRepositoryWizardIndex <> 0 Then
    (BorlandIDEServices As IOTAWizardServices).RemoveWizard(iRepositoryWizardIndex);
End.
```

Hope this provides helpful. We will use this building block for the next chapter where we'll create a project's code.

The code for this can be found attached to this PDF as `OTAChapter23ProjectRepositoryWizards.zip`.

24. OTA Template Wizard and Notifier Indexes...

This was originally published on 08 Nov 2011 using RAD Studio 2010.

I found a problem with my implementation with wizards and notifiers in add-ins the other day when I kept getting a crash on recompiling a Package in the IDE. It transpired that the problem was that I did not remove a compiler notifier on closing down the package so the IDE tried to use the notifier when it was not present. The reason for all of this was that the index returned by AddNotifier() returned a zero.

Now somewhere in the dim and distant past I thought I read that zero meant that the notifier was not install. It would seem that this was wrong and zero is a valid index. Having a look through all `AddNotifier()` methods in the `ToolsAPI.pas` module shows in some places that an error is indicated by the index return being < 0. Thus I've re-coded the installation and removal of wizards and notifier as follows.

First change the default starting index of the wizards and notifier to -1 as in the below code (this is now done through a constant):

```
Const
  iWizardFailState = -1;

Var
  ...
  {$ENDIF}
  iWizardIndex           : Integer = iWizardFailState;
  {$IFDEF D0006}
  iAboutPluginIndex      : Integer = iWizardFailState;
  {$ENDIF}
  iKeyBindingIndex       : Integer = iWizardFailState;
  iIDENotfierIndex       : Integer = iWizardFailState;
  {$IFDEF D2010}
  iCompilerIndex         : Integer = iWizardFailState;
  {$ENDIF}
  {$IFDEF D0006}
  iEditorIndex           : Integer = iWizardFailState;
  {$ENDIF}
  iRepositoryWizardIndex : Integer = iWizardFailState;
```

The code for installing the wizards and notifier has not changed but is included here for clarity:

```
Function InitialiseWizard(WizardType : TWizardType) : TWizardTemplate;

Var
  Svcs : IOTAServices;

Begin
  ...
  // Create Keyboard Binding Interface
  iKeyBindingIndex := (BorlandIDEServices As
      IOTAKeyboardServices).AddKeyboardBinding(
    TKeybindingTemplate.Create);
  // Create IDE Notifier Interface
  iIDENotfierIndex := (BorlandIDEServices As IOTAServices).AddNotifier(
    TIDENotifierTemplate.Create);
  {$IFDEF D2010}
  // Create Compiler Notifier Interface
  iCompilerIndex := (BorlandIDEServices As IOTACompileServices).AddNotifier(
    TCompilerNotifier.Create);
  {$ENDIF}
  {$IFDEF D2005}
  // Create Editor Notifier Interface
  iEditorIndex := (BorlandIDEServices As IOTAEditorServices).AddNotifier(
    TEditorNotifier.Create);
  {$ENDIF}
  // Create Project Repository Interface
```

```
  iRepositoryWizardIndex := (BorlandIDEServices As IOTAWizardServices).AddWizard(
    TRepositoryWizardInterface.Create);
End;
```

The last area of change is in the `Finalization` section where instead of checking for zero we check for -1 as follows:

```
Initialization
  ...
Finalization
  // Remove Wizard Interface
  If iWizardIndex > iWizardFailState Then
    (BorlandIDEServices As IOTAWizardServices).RemoveWizard(iWizardIndex);
  {$IFDEF D2005}
  // Remove Aboutbox Plugin Interface
  If iAboutPluginIndex > iWizardFailState Then
    (BorlandIDEServices As
      IOTAAboutBoxServices).RemovePluginInfo(iAboutPluginIndex);
  {$ENDIF}
  // Remove Keyboard Binding Interface
  If iKeyBindingIndex > iWizardFailState Then
    (BorlandIDEServices As
      IOTAKeyboardServices).RemoveKeyboardBinding(iKeyBindingIndex);
  // Remove IDE Notifier Interface
  If iIDENotfierIndex > iWizardFailState Then
    (BorlandIDEServices As IOTAServices).RemoveNotifier(iIDENotfierIndex);
  {$IFDEF D2010}
  // Remove Compiler Notifier Interface
  If iCompilerIndex <> iWizardFailState Then
    (BorlandIDEServices As IOTACompileServices).RemoveNotifier(iCompilerIndex);
  {$ENDIF}
  {$IFDEF D2005}
  // Remove Editor Notifier Interface
  If iEditorIndex <> iWizardFailState Then
    (BorlandIDEServices As IOTAEditorServices).RemoveNotifier(iEditorIndex);
  {$ENDIF}
  // Remove Repository Wizard Interface
  If iRepositoryWizardIndex <> iWizardFailState Then
    (BorlandIDEServices As IOTAWizardServices).RemoveWizard(iRepositoryWizardIndex);
End.
```

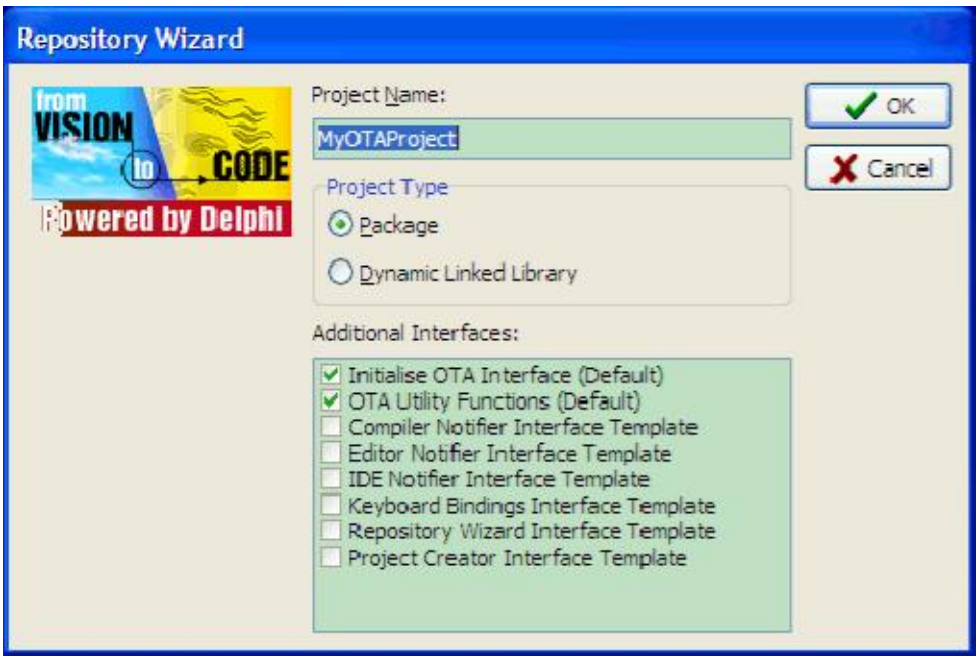
This bug would only have shown up in very rare circumstances but hopefully we can all learn from it.

25. Project creators

This was originally published on 10 Nov 2011 using RAD Studio 2010.

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 allows the user to select what they want to be created in a new project. We're 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 designed 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 in 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;
```

25.1 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 that 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;
```

25.2 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 return 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;
```

25.3 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}
```

25.4 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 defined 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 an 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 invoke 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.

Hope this is useful.

The code for this can be found attached to this PDF as `OTAChapter25ProjectCreators.zip`.

26. Unit Creators

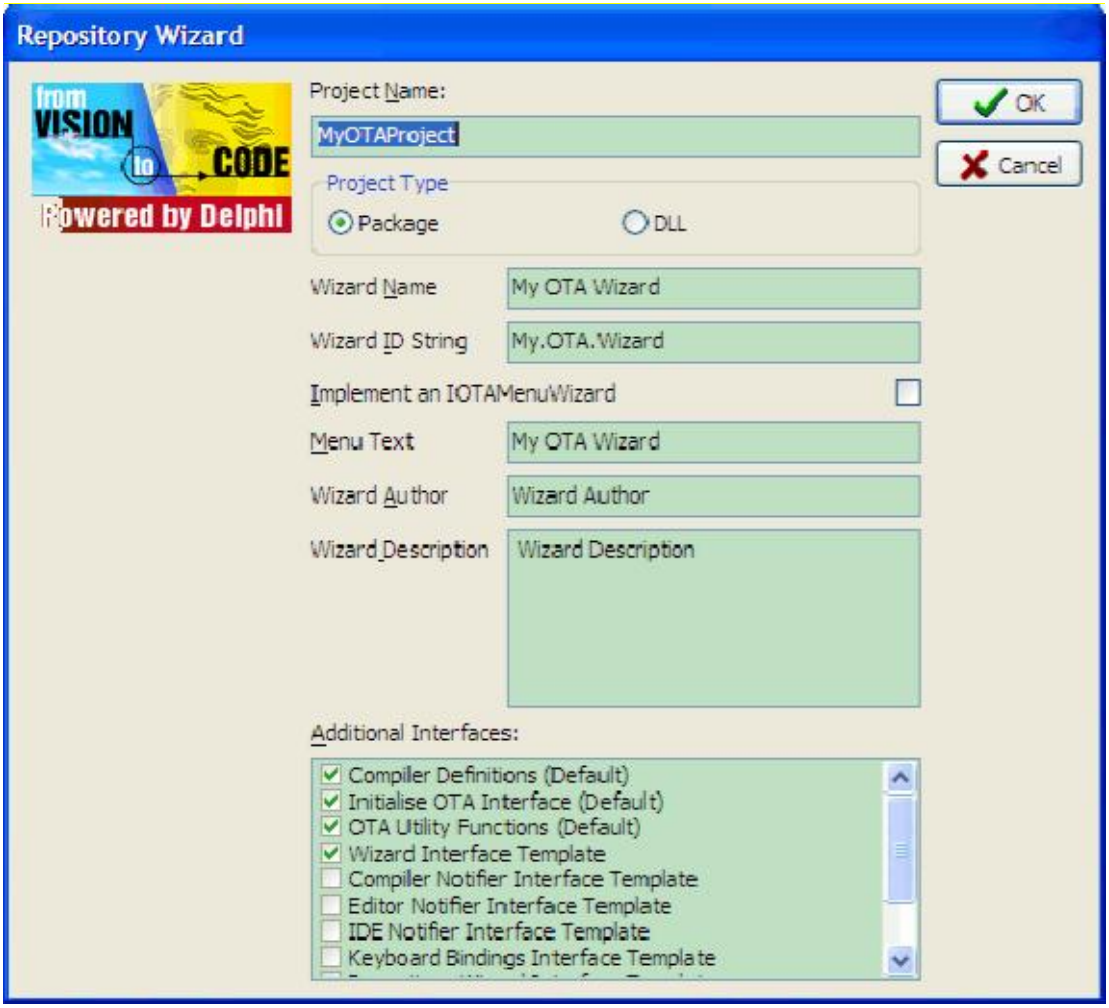
This was originally published on 16 Nov 2011 using RAD Studio 2010.

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.

26.1 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 := 'MyOTAProject';
      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;
  {$IFNDEF D2009}
  If edtProjectName.Text[1] In ['0'..'9'] Then
  {$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;
  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;
```

26.2 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')
  );
```

26.3 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 returns 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;
```

26.4 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 a where 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;
```

26.5 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]);
    {$IFNDEF D2009}
    SetLength(Result, Res.Size);
    Res.ReadBuffer(Result[1], Res.Size);
    {$ELSE}
    SetLength(strTemp, Res.Size);
    Res.ReadBuffer(strTemp[1], Res.Size);
    Result := String(strTemp);
    {$ENDIF}
  Finally
    Res.Free;
  End;
  Result := ExpandMacro(Result, '$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 allows 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 +
      '  iIDENotfierIndex := (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 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;
```

26.6 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. Enjoy :-)

The code for this can be found attached to this PDF as `OTAChapter26UnitCreators.zip`.

27. Open Tools API: Delphi 7 has documentation...

This was originally published on 14 Feb 2012 using RAD Studio 2010.

I stumbled on to something the other day that I didn't know about. Buried in the Delphi 7 (Personal for me) help directory is a help file for the Open Tools API. The contents page is broken and I cannot be sure of how extensive the help is but at 2 MB there should be (and on a brief look there is) a considerable amount of information in here.

Now, obviously, if you are using the newer features of the IDE then this will be no help but if you are coding your solutions for all version of Delphi down to 7 then this could be very useful for that core information.

28. IDE Main Menus

This was originally published on 28 Mar 2012 using RAD Studio 2010.

Well it's taken me a long time to produce this chapter because of what I believe are bugs in the various IDEs which I thought at the time was just me not doing it right.

The topic today is all about a more correct way of adding menus to the IDEs. The previous way I showed is a hangover from my very early days with the Open Tools API in Delphi 3 and does not take into account the changes that were introduced in later IDEs.

In producing this chapter I came across various problems but still wanted to produce a consistent way of adding menus to the IDE across all IDEs.

So let's start having a look at adding menus to the IDE.

Below is a definition of a class to manage the installation, lifetime and destruction of the menus.

```
  TApplicationMainMenu = Class
  {$IFDEF D2005} Strict {$ENDIF} Private
    FOTAMainMenu  : TMenuItem;
    {$IFNDEF D2005}
    FPatchTimer   : TTimer;
    {$ENDIF}
  {$IFDEF D2005} Strict {$ENDIF} Protected
    Procedure InstallMainMenu;
    Procedure AutoSaveOptionsExecute(Sender : TObject);
    Procedure AboutExecute(Sender : TObject);
    Procedure ProjCreateWizardExecute(Sender : TObject);
    Procedure ShowCompilerMessagesClick(Sender : TObject);
    Procedure ShowCompilerMessagesUpdate(Sender : TObject);
    Procedure ShowEditorMessagesClick(Sender : TObject);
    Procedure ShowEditorMessagesUpdate(Sender : TObject);
    Procedure ShowIDEMessagesClick(Sender : TObject);
    Procedure ShowIDEMessagesUpdate(Sender : TObject);
    {$IFNDEF D2005}
    Procedure PatchShortcuts(Sender : TObject);
    {$ENDIF}
  Public
    Constructor Create;
    Destructor  Destroy; Override;
  End;
```

The above class contains a number of `TNotifyEvents` for menu clicks and update event handlers for the actions however these are not the important items. The class contains an internal variable `FOTAMainMenu` to hold a reference to the main menu you create, such that freeing this menu will free all child menus and thus you don't need to hold reference to all the menus you add. Additionally, and for Delphi 7 and below, there is a timer that will patch the shortcut menus as the IDEs seem to lose this information. There is a method to install the menu, `InstallMainMenu` and a `PatchShortcuts` method for the Delphi 7 and below patching of shortcuts.

But first we need to understand how to create this class and subsequently your expert's main menu. To do this I've made an interval private variable for the class and created it in the Initialization section of the unit and freed it in the Finalization section of the unit. This way the menu does not need to be invoked by the main initialisation code where all your other experts are created but this does pose a problem. For those other elements to be able to be invoked by a menu they must expose a class method that invokes the functionality.

```
Var
  ApplicationMainMenu : TApplicationMainMenu;

Initialization
  ApplicationMainMenu := TApplicationMainMenu.Create;
Finalization
```

```
  ApplicationMainMenu.Free;
End.
```

The constructor below is fairly simple in that it initialises the menu reference to nil and runs the method `InstallMainMenu`. For Delphi 7 and below it also creates a `TTimer` control and assigns it to an event handler to patch the shortcuts.

```
constructor TApplicationMainMenu.Create;
begin
  FOTAMainMenu := Nil;
  InstallMainMenu;
  {$IFNDEF D2005} // Fixes a bug in D7 and below where shortcuts are lost
  FPatchTimer := TTimer.Create(Nil);
  FPatchTimer.Interval := 1000;
  FPatchTimer.OnTimer := PatchShortcuts;
  {$ENDIF}
end;
```

The destructor simply frees the menu reference (must be assigned in the `InstallMainMenu` method) and frees the timer in Delphi 7 and below.

```
destructor TApplicationMainMenu.Destroy;

begin
  {$IFNDEF D2005}
  FPatchTimer.Free;
  {$ENDIF}
  FOTAMainMenu.Free; // Frees all child menus
  Inherited Destroy;
end;
```

The `InstallMainMenu` method is where most of the work is done for creating the menus in the IDE. This relies on a number of utility methods which we will go through in a while but it's been designed to provide a simple interface for creating menus.

The below code checks that there is a main menu provided by the IDE and then creates a top level menu item assigning it to the `FOTAMainMenu` variable (so it can be freed later) and then creates the menu structure underneath that item.

You could use this technique to create a new menu structure underneath an existing IDE menu item but you will need to workout the menu item's name to do this.

I will describe the parameters of the `CreateMenuItem` method in a while.

```
procedure TApplicationMainMenu.InstallMainMenu;

Var
  NTAS : INTAServices;

begin
  NTAS := (BorlandIDEServices As INTAServices);
  If (NTAS <> Nil) And (NTAS.MainMenu <> Nil) Then
    Begin
      FOTAMainMenu := CreateMenuItem('OTATemplate', '&OTA Template', 'Tools',
        Nil, Nil, True, False, '');
      CreateMenuItem('OTAAutoSaveOptions', 'Auto Save &Option...', 'OTATemplate',
        AutoSaveOptionsExecute, Nil, False, True, 'Ctrl+Shift+O');
      CreateMenuItem('OTAProjectCreatorWizard', '&Project Creator Wizard...',
        'OTATemplate', ProjCreateWizardExecute, Nil, False, True, 'Ctrl+Shift+P');
      CreateMenuItem('OTANotifiers', 'Notifer Messages', 'OTATemplate', Nil, Nil,
        False, True, '');
      CreateMenuItem('OTAShowCompilerMsgs', 'Show &Compiler Messages',
        'OTANotifiers', ShowCompilerMessagesClick, ShowCompilerMessagesUpdate,
        False, True, '');
      CreateMenuItem('OTAShowEditorrMsgs', 'Show &Editor Messages',
        'OTANotifiers', ShowEditorMessagesClick, ShowEditorMessagesUpdate,
```

```
        False, True, '');
      CreateMenuItem('OTAShowIDEMsgs', 'Show &IDE Messages',
        'OTANotifiers', ShowIDEMessagesClick, ShowIDEMessagesUpdate,
        False, True, '');
      CreateMenuItem('OTASeparator0001', '', 'OTATemplate', Nil, Nil, False, True,
       '');
      CreateMenuitem('OTAAbout', '&About...', 'OTATemplate', AboutExecute, Nil,
        False, True, 'Ctrl+Shift+Q');
    End;
end;
```

Below are examples of `OnClick` and `OnUpdate` event handlers for the actions associated with the menus. Here I've used an enumerate and set to handle some options in the application and update the checked property of the action based on the inclusion or exclusion of the enumerate in the set. The click action simply adds or removes the enumerate from the set. You will probably ask why I don't use include or exclude for the sets and enumerates. Since the set is a property of a class, you cannot use the include or exclude methods on a property of a class.

```
Procedure UpdateModuleOps(Op : TModuleOption);

Var
  AppOps : TApplicationOptions;

Begin
  AppOps := ApplicationOps;
  If Op In AppOps.ModuleOps Then
    AppOps.ModuleOps := AppOps.ModuleOps - [Op]
  Else
    AppOps.ModuleOps := AppOps.ModuleOps + [Op];
End;

procedure TApplicationMainMenu.ShowCompilerMessagesClick(Sender: TObject);
begin
  UpdateModuleOps(moShowCompilerMessages);
end;

procedure TApplicationMainMenu.ShowCompilerMessagesUpdate(Sender: TObject);
begin
  If Sender Is TAction Then
    With Sender As TAction Do
      Checked := moShowCompilerMessages In ApplicationOps.ModuleOps;
end;
```

Finally for this module we have the Delphi 7 `OnTimer` event handler for the patching of the shortcuts. This is handled by a utility function which we will look at in a while but the event waits for a visible IDE before invoking the utility function and then switches off the timer.

```
{$IFNDEF D2005}
Procedure TApplicationMainMenu.PatchShortcuts(Sender : TObject);

Begin
  If Application.MainForm.Visible Then
    Begin
      PatchActionShortcuts(Sender);
      FPatchTimer.Enabled := False;
    End;
End;
{$ENDIF}
```

Now for the utility functions.

The `AddImageToIDE` function is called internally by `CreateMenuItem` and adds an image from the projects resource file to the IDEs image list and returns the index of that image in the IDEs image list so that it can be referenced in the action. You will note that there is a commented out section of this method, this is because it continually caused an exception, so an older method is used. If the resource is not found then no image is added to the IDE and -1 is returned as the image index.

```
Function AddImageToIDE(strImageName : String) : Integer;

Var
  NTAS : INTAServices;
  ilImages : TImageList;
  BM : TBitMap;

begin
  Result := -1;
  If FindResource(hInstance, PChar(strImageName + 'Image'), RT_BITMAP) > 0 Then
    Begin
      NTAS := (BorlandIDEServices As INTAServices);
      // Create image in IDE image list
      ilImages := TImageList.Create(Nil);
      Try
        BM := TBitMap.Create;
        Try
          BM.LoadFromResourceName(hInstance, strImageName + 'Image');
          {$IFDEF D2005}
          ilImages.AddMasked(BM, clLime);
          // EXCEPTION: Operation not allowed on sorted list
          // Result := NTAS.AddImages(ilImages, 'OTATemplateImages');
          Result := NTAS.AddImages(ilImages);
          {$ELSE}
          Result := NTAS.AddMasked(BM, clLime);;
          {$ENDIF}
        Finally
          BM.Free;
        End;
      Finally
        ilImages.Free;
      End;
    End;
end;
```

The `FindMenuItem` function is called internally by `CreateMenuItem` and is used to find a named menu item (i.e. the name assigned to the name property of an existing menu item. The named menu item is returned if found else `nil` is returned. This function recursively searches the main menu system.

```
function FindMenuItem(strParentMenu : String): TMenuItem;

  Function IterateSubMenus(Menu : TMenuItem) : TMenuItem;

  Var
    iSubMenu : Integer;

  Begin
    Result := Nil;
    For iSubMenu := 0 To Menu.Count - 1 Do
      Begin
        If CompareText(strParentMenu, Menu[iSubMenu].Name) = 0 Then
          Result := Menu[iSubMenu]
        Else
          Result := IterateSubMenus(Menu[iSubMenu]);
        If Result <> Nil Then
          Break;
      End;
  End;

Var
  iMenu : Integer;
  NTAS : INTAServices;
  Items : TMenuItem;

begin
  Result := Nil;
  NTAS := (BorlandIDEServices As INTAServices);
  For iMenu := 0 To NTAS.MainMenu.Items.Count - 1 Do
```

```
    Begin
      Items := NTAS.MainMenu.Items;
      If CompareText(strParentMenu, Items[iMenu].Name) = 0 Then
        Result := Items[iMenu]
      Else
        Result := IterateSubMenus(Items);
      If Result <> Nil Then
        Break;
    End;
end;
```

This next method is the heart of the experts ability to create a menu item in the IDEs main menu system and I will explain how it works.

- Firstly an image is added to the IDEs image list (if the resource exists in the expert);
- Next the menu item is created with the main menu as its owner;
- Next, if there is an `OnClick` event handler, then an Action is created and assigned various attributes like caption, etc;
- Next a catch is made for menu items that have no event handler (heads of sub-menus or separators);
- Then the action is assigned to the menu;
- This position of the parent menu is located;
- Adds the menu to the IDE relative to the parent menu.

You will probably note that there is more commented out code, this is because the “new” way to create menus in the IDE does not create icons next to the menus. It could be something that I'm not doing right but I spent an inordinate amount of time trying to get it to work.

Some explanation of the parameter is also needed as follows:

- `strName` – This is the name of the action / menu (which will be appropriately appended with text);
- `strCaption` – This is the name (with accelerator) of the action / menu;
- `strParentMenu` – This is the name of the parent menu. This is either the menu under which you want child menus or is the menu item which comes before or after your new menu depending on the below options;
- `ClickProc` – This is the `OnClick` event handler for the action / menu that does something when the menu or action is clicked or invoked. If you do not want to implement this, say for a top level menu, the pass nil;
- `UpdateProc` – This is an optional `OnUpdate` event handler for the action /menu. If you do not want to implement this simply pass nil;
- `boolBefore` – If true this will make the new menu appear before the Parent menu item;
- `boolChildMenu` – If true this will add the new menu as a child of the Parent menu;
- `strShortcut` – This is a shortcut string to be assigned to the action / menu. Just pass an empty string if you do not want to implement a shortcut.

```
Function CreateMenuItem(strName, strCaption, strParentMenu : String;
  ClickProc, UpdateProc : TNotifyEvent; boolBefore, boolChildMenu : Boolean;
  strShortCut : String) : TMenuItem;

Var
  NTAS : INTAServices;
  CA : TAction;
  //{$IFNDEF D2005}
  miMenuItem : TMenuItem;
  //{$ENDIF}
  iImageIndex : Integer;

begin
  NTAS := (BorlandIDEServices As INTAServices);
  // Add Image to IDE
```

```
  iImageIndex := AddImageToIDE(strName);
  // Create the IDE action (cached for removal later)
  CA := Nil;
  Result := TMenuItem.Create(NTAS.MainMenu);
  If Assigned(ClickProc) Then
    Begin
      CA := TAction.Create(NTAS.ActionList);
      CA.ActionList := NTAS.ActionList;
      CA.Name := strName + 'Action';
      CA.Caption := strCaption;
      CA.OnExecute := ClickProc;
      CA.OnUpdate := UpdateProc;
      CA.ShortCut := TextToShortCut(strShortCut);
      CA.Tag := TextToShortCut(strShortCut);
      CA.ImageIndex := iImageIndex;
      CA.Category := 'OTATemplateMenus';
      FOTAActions.Add(CA);
    End Else
  If strCaption <> '' Then
    Begin
      Result.Caption := strCaption;
      Result.ShortCut := TextToShortCut(strShortCut);
      Result.ImageIndex := iImageIndex;
    End Else
      Result.Caption := '-';
  // Create menu (removed through parent menu)
  Result.Action := CA;
  Result.Name := strName + 'Menu';
  // Create Action and Menu.
  //{$IFDEF D2005}
  // This is the new way to do it BUT doesnt create icons for the menu.
  //NTAS.AddActionMenu(strParentMenu + 'Menu', CA, Result, boolBefore,
      boolChildMenu);
  //{$ELSE}
  miMenuItem := FindMenuItem(strParentMenu + 'Menu');
  If miMenuItem <> Nil Then
    Begin
      If Not boolChildMenu Then
        Begin
          If boolBefore Then
            miMenuItem.Parent.Insert(miMenuItem.MenuIndex, Result)
          Else
            miMenuItem.Parent.Insert(miMenuItem.MenuIndex + 1, Result);
        End Else
          miMenuItem.Add(Result);
    End;
  //{$ENDIF}
end;
```

This next utility function is used to patch the IDE shortcuts which are lost by Delphi 7 and below. This method is called by the on timer event handler in the `MainMenuModule`. It uses the fact that we stored the menu shortcut in the tag property to re-apply the shortcut after the IDE is loaded.

```
Procedure PatchActionShortcuts(Sender : TObject);

Var
  iAction : Integer;
  A : TAction;

Begin
  For iAction := 0 To FOTAActions.Count - 1 Do
    Begin
      A := FOTAActions[iAction] As TAction;
      A.ShortCut := A.Tag;
    End;
End;
```

Finally, this last utility function is provided to remove any of your custom actions from the toolbars. If you unloaded your BPL file and then tried to use your custom action then you would have an access violation in the IDE.

```
Procedure RemoveToolbarButtonsAssociatedWithActions;

  Function IsCustomAction(Action : TBasicAction) : Boolean;

  Var
    i: Integer;

  Begin
    Result := False;
    For i := 0 To FOTAActions.Count - 1 Do
      If Action = FOTAActions[i] Then
        Begin
          Result := True;
          Break;
        End;
  End;

  Procedure RemoveAction(TB : TToolbar);

  Var
    i: Integer;

  Begin
    If TB <> Nil Then
      For i := TB.ButtonCount - 1 DownTo 0 Do
        Begin
          If IsCustomAction(TB.Buttons[i].Action) Then
            TB.RemoveControl(TB.Buttons[i]);
        End;
  End;

Var
  NTAS : INTAServices;

Begin
  NTAS := (BorlandIDEServices As INTAServices);
  RemoveAction(NTAS.ToolBar[sCustomToolBar]);
  RemoveAction(NTAS.Toolbar[sStandardToolBar]);
  RemoveAction(NTAS.Toolbar[sDebugToolBar]);
  RemoveAction(NTAS.Toolbar[sViewToolBar]);
  RemoveAction(NTAS.Toolbar[sDesktopToolBar]);
  {$IFDEF D0006}
  RemoveAction(NTAS.Toolbar[sInternetToolBar]);
  RemoveAction(NTAS.Toolbar[sCORBAToolBar]);
  {$IFDEF D2009}
  RemoveAction(NTAS.Toolbar[sAlignToolbar]);
  RemoveAction(NTAS.Toolbar[sBrowserToolbar]);
  RemoveAction(NTAS.Toolbar[sHTMLDesignToolbar]);
  RemoveAction(NTAS.Toolbar[sHTMLFormatToolbar]);
  RemoveAction(NTAS.Toolbar[sHTMLTableToolbar]);
  RemoveAction(NTAS.Toolbar[sPersonalityToolBar]);
  RemoveAction(NTAS.Toolbar[sPositionToolbar]);
  RemoveAction(NTAS.Toolbar[sSpacingToolbar]);
  {$ENDIF}
  {$ENDIF}
End;
```

The last part of the utility unit creates and frees the memory for a collection holding the actions that we've added to the IDE. Since the collection owns the action, freeing the collection removes the action from the IDE.

```
Initialization
  FOTAActions := TObjectList.Create(True);
Finalization
```

```
  RemoveToolbarButtonsAssociatedWithActions;
  FOTAActions.Free;
End.
```

Well I hope that's useful. In the next couple of chapters I'm going to look at creating forms and inherited forms.

The code for this can be found attached to this PDF as `OTAChapter28IDEMenus.zip`.

29. Creating Forms

This chapter is new to this book and uses Delphi XE7.

This chapter will talk about creating forms using the Open Tools API based on the knowledge from our chapter on Unit Creators.

There are two ways we could create forms using the Open Tools API: the first is to provide the `.pas` and `.dfm` source files and the second is to add controls to the form after basic forms have been created. I'll talk about each.

29.1 Forms from Source Code

From our previous chapter we would need to makes some changes to our code to create a form.

The first change would be in the `GetCreateType` method of our `TModuleCreator` class (implements the `IOTACreator` interface) where we would need to return `sForm` instead of `sUnit`.

The second change would be to return an `IOTAFile` interface from the `NewFormFile` method of the `TModuleCreator` (implements the `IOTAModuleCreator` interface) to provide the *DFM* file code information for the class. This method provides two parameters which should be passed to your `IOTAFile` implementation: one for the form name and the second for the form's inheritance identifier.

Finally the `NewImplSource` method would need to be updated to provide the implementation code for the form class.

It should go without saying that obviously you need to ensure that the *DFM* and *PAS* code you provide are synchronised and valid definitions for a form containing any controls you want on the forms and the DFM files contains their published property settings required.

29.2 Forms from Adding Controls

For the alternative way to provide a form you would still have to provide the *PAS* and *DFM* implementation however these can be plain `TForm` implementations.

To add the controls to the form we would need to implement code in the `FormCreated` method of the `IOTAModuleCreator`. This method provides access to the form through an implementation of `IOTAFormEditor`.

The definition of the `IOTAFormEditor` (from XE7) is as follows:

```
IOTAFormEditor = interface(IOTAEditor)
  ['{F17A7BD2-E07D-11D1-AB0B-00C04FB16FB3}']
  function GetRootComponent: IOTAComponent;
  function FindComponent(const Name: string): IOTAComponent;
  function GetComponentFromHandle(ComponentHandle: TOTAHandle): IOTAComponent;
  function GetSelCount: Integer;
  function GetSelComponent(Index: Integer): IOTAComponent;
  function GetCreateParent: IOTAComponent;
  function CreateComponent(const Container: IOTAComponent;
    const TypeName: string; X, Y, W, H: Integer): IOTAComponent;
  procedure GetFormResource(const Stream: IStream);
end;
```

Unfortunately there are no comments with the code so we will have to try and understand the methods from their names and parameters. Most of the interface method either require parameters of the type `IOTAComponent` or return an `IOTAComponent` so it would be useful to have a look at this interface before discussing the `IOTAFormEditor` methods.

29.2.1 The IOTAComponent Interface

This interface has the following definition (from XE7).

```
IOTAComponent = interface(IUnknown)
  ['{AC139ADF-329A-D411-87C6-9B2730412200}']
  function GetComponentType: string;
  function GetComponentHandle: TOTAHandle;
  function GetParent: IOTAComponent;
  function IsTControl: Boolean;
  function GetPropCount: Integer;
  function GetPropName(Index: Integer): string;
  function GetPropType(Index: Integer): TTypeKind;
  function GetPropTypeByName(const Name: string): TTypeKind;
  function GetPropValue(Index: Integer; var Value): Boolean;
  function GetPropValueByName(const Name: string; var Value): Boolean;
  function SetProp(Index: Integer; const Value): Boolean;
  function SetPropByName(const Name: string; const Value): Boolean;
  function GetChildren(Param: Pointer; Proc: TOTAGetChildCallback): Boolean;
  function GetControlCount: Integer;
  function GetControl(Index: Integer): IOTAComponent;
  function GetComponentCount: Integer;
  function GetComponent(Index: Integer): IOTAComponent;
  function Select(AddToSelection: Boolean): Boolean;
  function Focus(AddToSelection: Boolean): Boolean;
  function Delete: Boolean;
  //function GetIPersistent: IPersistent;
  //function GetIComponent: IComponent;
end;
```

GetComponentType

From the comments provided this method returns a string representing the type of the component that the `IOTAComponent` references.

GetComponentHandle

From the comments provided this method returns a unique Handle to the `TComponent` / `TPersistent` referenced by the `IOTAComponent`.

GetParent

From the comments provided this method returns the interface corresponding to the parent control if a `TControl`, otherwise returns the owner of the control. If it's a `TPersistent` or the root object then it returns `Nil`.

IsTControl

From the comments provided this method returns `True` if component is a `TControl` descendant,

GetPropCount

From the comments provided this method returns the number of published properties on this component.

GetPropName

From the comments provided this method, given the index of the property of the component the method returns the property name.

GetPropType

From the comments provided this method, given the index of the property of the component the method returns the property type. Note this is a *RTTI* enumerate.

GetPropTypeByName

From the comments provided this method, given the name of the property of the component the method returns the property type (an *RTTI* enumerate as above).

GetPropValue & GetPropValueByName

From the comments provided this method and given the index or name of the property of the component, this method returns the property value. The untyped `var` parameter must be large enough to hold the returned value. For string types, the untyped `var` parameter must match the actual string type (as indicated by `GetPropType` or `GetPropTypeByName`):

- `tkString`: the untyped parameter should be of type `ShortString`;
- `tkLString`: the untyped parameter should be of type `AnsiString`;
- `tkWString`: the untyped parameter should be of type `WideString`;
- `tkUString`: the untyped parameter should be of type `UnicodeString`.

If the property is a descendant of `TPersistent`, the return value is an `IOTAComponent`. For properties of any other objecttype, the return value is `Nil`.

SetProp & SetPropByName

From the comments provided this method, given the index or name of the property of the component sets the property value. It should go without saying that the constant value should of the same type as the property being set.

GetChildren

From the comments provided this method enumerates the child controls just like `TComponent.GetChildren`. Therefore for each child component of the current component the method `TOTAGetChildCallback` will be called (you need to define this). The pointer parameter is passed to the call back for each component so you can use this to pass one piece of information to the callback.

GetControlCount

From the comments provided this method returns the number of child controls (if the component is a `TWinControl` / `TWidgetControl` descendant, else it returns 0).

GetControl

From the comments provided this method, given the index of the child control on the current component, returns an interface to the child control.

GetComponentCount

From the comments provided this method returns the number of child components (if the control is a `TComponent` descendant, else it returns 0).

GetComponent

From the comments provided this method, given the index of the component on the current control, returns an interface to the child component.

Select

From the comments provided this method selects the current component and updates the Object Inspector. If `AddToSelection` is `True`, then the current selection is not cleared, and the components are multi-selected.

Focus

From the comments provided this method is the same as `Select` except it brings the form to the front with the component selected. If this interface is a Form / Data Module, then `Focus` only brings the form to front. See `Select` for description of the `AddToSelection` parameter.

Delete

From the comments provided this method deletes the component from the form. Following this call, this interface will now be invalid and must be release.

GetIPersistent

From the comments provided this method is no longer in use but it did returns the `IPersistent` interface for the component.

GetIComponent

From the comments provided this method is no longer in use but it did returns the `IComponent` interface if instance is a `TComponent` else `Nil`.

29.2.2 IOTAFormEditor Methods

Since there is only one comment in this definition I'll interpret the meaning of the method from its name and also what we know about the `IOTAComponent` interface above.

GetRootComponent

From the comment in the definition this returns the form editor root component. I am assuming that this is how to get an `IOTAComponent` reference for the form and thus allow you to use the `IOTAComponent` interface to do most of your form management.

FindComponent

This method returns the `IOTAComponent` interface for the named component (i.e. the component with the name property identified). What is not known where it whether an exception is raised is the named component does not exist or whether `Nil` is returned – you will have to test this.

GetComponentFromHandle

This method takes an `TOTAHandle` (a pointer) to return a reference to an `IOTAComponent`. As above, it's not clear what the status of the returned value will be if the handle is not found or whether an exception is raised.

GetSelCount

This method it is assumed returns the number of components that are currently selected on the form.

GetSelComponent

This method returns an `IOTAComponent` reference for the indexed selected component (the index is probably between `0` and `GetSelCount - 1`.

GetCreateParent

I assume that this method returns the `IOTAComponent` interface that should be used as the parent for creating new components.

CreateComponent

This method is where you create new components. I should be stated here that the IDE in which the OTA expert is running MUST have the appropriate BPL of components loaded in order for the IDE to add a component to the form else I'm sure an exception will be raised.

The parameters seem obvious as follows:

- `Container`: This is the container (parent) in which the new component is to be created and also defines the `X`, `Y`, `W`, and `H` coordinate below;
- `TypeName`: This is the type name for the new component, for instance `TButton`;
- `X`: This is the horizontal position (within the parent) from the left for the top left corner of the new component;
- `Y`: This is the vertical position (within the parent) from the top for the top left corner of the new component;
- `W`: This is the width of the component;
- `H`: This is the height of the component.

GetFormResource

This method would seem to provide the Steam reference for the form's *RES* file. Really not sure of the format but would think that it would be a binary stream for a windows resource.

I hope this provides you with enough information to create forms in your OTA experts.

30. Shared Units in OTA Projects

This chapter is new to this book.

In this chapter I want to talk through shared modules in OTA projects and the caveats of using them.

Put simply you cannot load the same module into the IDE twice in BPL format! So for instance in my projects I have a library module called `DGHLibrary.pas` which contains a load of library code I've written over the years to do things that I regularly do in my projects. This means that I cannot use this module in more than one BPL styles OTA project loaded into the IDE.

So how do we get around this limitation? The way I do this is to create DLL version of my OTA projects only as this limitation do not apply with these styles of OTA project but what do you do if the absolutely need to have your OTA project in the BPL format?

The answer to the above question is that all shared code, whether its library code as I've described above or your own components needs to be built into a shared BPL file and loaded into the IDE before loading the OTA Project.

Similarly this will apply to the creation of BPL OTA projects that reference components in the component pallet where you will need to ensure that the BPL OTA project is built with the appropriate run-time packages.

31. The Trials and Tribulations of upgrading to a new IDE

This chapter is new to this book and used Delphi XE7.

I always dread upgrading IDEs as there can be and has been in the pass a lot of work needed to update your OTA Projects to get them to work. Embarcadero/CodeGear/Borland have historically broken parts of the IDE's OTA and / or implemented a different method and depreciating an existing method for doing things requiring you to re-write portions of your code.

Here I go through how I tackle the upgrade and what I found when upgrading the OTA Template code from Delphi XE2 to XE7.

31.1 Creating a new IDE project for an existing OTA Project

If the OTA project you want to upgrade is for just yourselves and you only need it to work in the new IDE then you can simply open the project in the new IDE (which may upgrade the `.dproj` file information), change the projects name and re-compile. However if like me you want to maintain version of the OTA project for older IDE this is not advised as it may change your `.dproj` file so that it is incompatible with earlier versions of the IDE.

What I do is copy the DPR project and rename it in-line with the new IDE, so for instance `OTATemplateXE2.dpr` is copied and renamed to `OTATemplateXE7.dpr`. I then open this project in the new IDE and go straight to the projects options as these need to be updated.

There are two options I specifically want to set for the new project and these are the output directory for the DLL/BPL which is usually one directory higher in my filing structure (`..\`) and the unit output directory where all the DCU files will go (`..\XE7DCUs`). This second setting is important if you maintain multiple version of the OTA project as the DCU file will be incompatible between versions therefore I have different DCU directories for different compilers.

31.2 Updating any Conditional Definitions

Once the above was done trying to compile the project raised a number syntax errors which needed to be addressed. Have not upgraded IDEs for at least 2 ½ years it took me a few minutes to understand why these were occurring.

All the errors stemmed from code for older IDEs not being compatible for the new IDE and I realised that I needed to update my conditional compilation include file by adding the below definitions to the file to account for Delphi XE3 through XE7.

```
{$IFDEF VER240}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE00}
  {$DEFINE DXE20}
  {$DEFINE DXE30}
{$ENDIF}

{$IFDEF VER250}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
```

```
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE00}
  {$DEFINE DXE20}
  {$DEFINE DXE30}
  {$DEFINE DXE40}
{$ENDIF}

{$IFDEF VER260}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE00}
  {$DEFINE DXE20}
  {$DEFINE DXE30}
  {$DEFINE DXE40}
  {$DEFINE DXE50}
{$ENDIF}

{$IFDEF VER270}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE00}
  {$DEFINE DXE20}
  {$DEFINE DXE30}
  {$DEFINE DXE40}
  {$DEFINE DXE50}
  {$DEFINE DXE60}
{$ENDIF}

{$IFDEF VER280}
  {$DEFINE D0002}
  {$DEFINE D0003}
  {$DEFINE D0004}
  {$DEFINE D0005}
  {$DEFINE D0006}
  {$DEFINE D0007}
  {$DEFINE D0008}
  {$DEFINE D2005}
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE00}
  {$DEFINE DXE20}
```

```
  {$DEFINE DXE30}
  {$DEFINE DXE40}
  {$DEFINE DXE50}
  {$DEFINE DXE60}
  {$DEFINE DXE70}
{$ENDIF}
```

31.3 Setting up the project for Debugging

You might think that once the project has compiled all is good but please do not assume this. There can be problems with new IDEs that don't exist in older IDEs due to Embarcadero changing the internals of Delphi but also these errors may not present themselves without you debugging the project in another IDE as the IDE can suppress errors on close down.

To set the parameters of the OTA project to load it into another version of the same IDE but using the –r command line parameter which creates a new registry key for that instance of the IDE allowing you to have a vanilla instance of the IDE. So for instance I used the below parameters to the BDS.exe.

```
-r OTATemplateXE7DLL
```

31.4 Bugs found in the OTA Template code for XE7

Well as alluded to above I found a number of issues with the OTA Template code I had written for Delphi 2010. These errors manifested themselves as access violations when the IDE was closing down. These didn't existing in the older IDEs so I can only presume that Embarcadero have changes some internal implementations.

The first pair of errors that were encountered were where I was clearing the messages from the messages window in the `Finalization` section of 2 modules, the IDE notifier module and the repository wizard module. I assume for some reason that these interfaces are not valid any longer when the IDE is closing down. So as shown below the offending lines of code are deleted from the `Finalization` section so the code and moved to the `Destroy` methods of the appropriate wizard class.

The below line should be cut from the `Finalization` section.

```
Initialization
Finalization
  ClearMessages([cmCompiler..cmTool]); // DELETE
End.
```

The above cut line should be pasted into the below `Destroy` method.

```
Procedure TIDENotifierTemplate.Destroyed;

Begin
  ClearMessages([cmCompiler..cmTool]); // ADDED
  If moShowIDEMessages In ApplicationOps.ModuleOps Then
    OutputMessage('Destroyed' {$IFDEF D0006}, strIDENotifierMessages {$ENDIF});
End;
```

The below line should be cut from the Finalization section.

```
Initialization
  FProjWizardRef := Nil;
Finalization
  FProjWizardRef := Nil;
  ClearMessages([cmCompiler..cmTool]); // DELETE
End.
```

The above cut line should be pasted into the below Destroy method.

```
Procedure TRepositoryWizardInterface.Destroyed;

```

```
Begin
  ClearMessages([cmCompiler..cmTool]); // ADD
  OutputMessage('Destroyed' {$IFDEF D0006}, strRepositoryWizardGroup {$ENDIF});
End;
```

The second error was of a similar nature and was caused by the IDE notifier wizard trying to add messages to the IDE's message window when the IDE was closing down. To solve this I've added an extra line of code to the below utility method to only output messages if the IDE is visible.

```
Procedure OutputMessage(strText : String; strGroupName : String);

Var
  Group : IOTAMessageGroup;

Begin
  If Application.MainForm.Visible Then // ADDED
    With (BorlandIDEServices As IOTAMessageServices) Do
      Begin
        Group := GetGroup(strGroupName);
        If Group = Nil Then
          Group := AddMessageGroup(strGroupName);
        AddTitleMessage(strText, Group);
      End;
End;
```

31.5 Debugging BPL files in XE7

I have to admit that I'm stumped on the next issue as I think from the testing I have done that there is a serious issue with the IDE.

After debugging the above OTA project in DLL format I found that trying to debug the BPL version of the file caused the main IDE once the secondary one had closed to simply just disappear and a web browser appear with a message about licensing issues with the IDE. I found that commenting all the code out and just loading the BPL file was okay but as soon as I even declared a wizard interface (not even creating an instance) caused the main IDE to disappear.

A number of my existing projects are fine for debugging including Eidolon, an Excel COM DLL and my Folder Sync application, along with an application to syntax highlight the code in this Word file.

32. Dockable Forms

This chapter is new to this book.

This is an interesting topic and one I have to admit needing help on when I originally coded this by looking at the code used in GExperts. Allen Bauer wrote an article on dockable forms a long time ago (Opening Doors: Getting Inside the IDE - by Allen Bauer) but only showed how to create them but omitted the relative information needed to allow the form to persist in the IDE.

32.1 Defining a Dockable Form

The first part of the creating a dockable form is to create a new form in your OTA project. In the examples below I'm using code from my *Browse and Doc It* OTA project. The form is an empty form with no components on it as the browser is contained in a frame which is later in the loading process inserted into the dockable form. This is done as the form need to be created and made available to the IDE at a different point in time from the wizard which creates the browser (if you read Allan's article above it states that the IDE's desktop mechanism needs to be able to instantiate your dockable form which is why the following code is not created and destroyed in the same manner as a wizard).

```
object frmDockableModuleExplorer: TfrmDockableModuleExplorer
  Left = 456
  Top = 303
  Caption = 'Module Explorer'
  ClientHeight = 358
  ClientWidth = 257
  Color = clBtnFace
  Constraints.MinHeight = 250
  Constraints.MinWidth = 100
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  KeyPreview = True
  OldCreateOrder = False
  ShowHint = True
  PixelsPerInch = 96
  TextHeight = 13
end
```

The below code shows the declaration of the form class. You should note that the form need to be inherited from an IDE internal form `TDockableForm` which requires the `DockForm` unit to be added to the `uses` clause.

The below class employs a number of class method that allow it to be interacted with as a singleton pattern. Not all of these are required to create a dockable form but do show you how I manage mine.

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DockForm, ModuleExplorerFrame, BaseLanguageModule;

type
  TfrmDockableModuleExplorer = class(TDockableForm)
  private
    FModuleExplorerFrame : TframeModuleExplorer;
  public
    Constructor Create(AOwner : TComponent); Override;
    Destructor Destroy; Override;
    Procedure Focus;
    Class Procedure ShowDockableModuleExplorer;
    Class Procedure RemoveDockableModuleExplorer;
    Class Procedure CreateDockableModuleExplorer;
    Class Procedure RenderDocumentTree(BaseLanguageModule : TBaseLanguageModule);
    Class Procedure HookEventHandlers(SelectionChangeProc : TSelectionChange;
      Focus, ScopeChange : TNotifyEvent);
```

```
  end;
```

We also need to declare another type (Class of Class) for one of our methods below.

```
  TfrmDockableModuleExplorerClass = Class of TfrmDockableModuleExplorer;
```

Now we come to the implementation of the code. Firstly we need to add a new unit to the implementation uses clause to allow us to load and save desktop information. Secondly we need to declare a private module variable to hold the singleton form instance for the dockable form.

```
implementation

{$R *.dfm}

Uses
  DeskUtil;

Var
  FormInstance : TfrmDockableModuleExplorer;
```

The below `RegisterDockableForm` is called when the `CreateDockableForm` method is called. This method registers the custom dockable form with the IDE so that it knows about the form can load and save the form's position and docking when the desktop files are loaded and saved.

```
Procedure RegisterDockableForm(FormClass : TfrmDockableModuleExplorerClass;
  var FormVar; Const FormName : String);

Begin
  If @RegisterFieldAddress <> Nil Then
    RegisterFieldAddress(FormName, @FormVar);
  RegisterDesktopFormClass(FormClass, FormName, FormName);
End;
```

Similarly the below `UnregisterDockableForm` is called when the `FreeDockableForm` method is called. This method unregisters the custom dockable form with the IDE so that it no longer knows about the form and therefore does not load and save the form's position and docking when the desktop files are loaded and saved.

```
Procedure UnRegisterDockableForm(var FormVar; Const FormName : String);

Begin
  If @UnRegisterFieldAddress <> Nil Then
    UnregisterFieldAddress(@FormVar);
End;
```

This method `CreateDockableForm` is called by `CreateDockableModuleExplorer` if the form does not already exist to create the form and register it with the IDE so that its position and docking are persistent.

```
Procedure CreateDockableForm(var FormVar : TfrmDockableModuleExplorer;
  FormClass : TfrmDockableModuleExplorerClass);

Begin
  TCustomForm(FormVar) := FormClass.Create(Nil);
  RegisterDockableform(FormClass, FormVar, TCustomForm(FormVar).Name);
End;
```

This method `FreeDockableForm` is called by `RemoveDockableModuleExplorer`. If the form exists it unregisters the form from the IDE and free's the memory used by the form. It's important here that the form reference is set to `Nil` using `FreeAndNil`.

```
Procedure FreeDockableForm(var FormVar : TfrmDockableModuleExplorer);

Begin
```

```
  If Assigned(FormVar) Then
    Begin
      UnRegisterDockableForm(FormVar, FormVar.Name);
      FreeAndNil(FormVar);
    End;
End;
```

This method ShowDockableForm is called by ShowDockableModuleExplorer. If the form does not exist this method exits doing nothing. If the form is floating (not docked) the form is shown and focused (bought to the front from behind other windows). If the Form is docked then the form is shown (it could be hidden as a tab) and focused.

```
Procedure ShowDockableForm(Form : TfrmDockableModuleExplorer);

Begin
  If Not Assigned(Form) Then
    Exit;
  If Not Form.Floating Then
    Begin
      Form.ForceShow;
      FocusWindow(Form);
      Form.Focus;
    End Else
    Begin
      Form.Show;
      Form.Focus;
    End;
End;
```

This method creates the dockable form. Since I didn't want to mess around with form inheritance at the time the few control on the form are created in code (a la TurboVision). The name of the desktop section in the desktop files is assigned and AutoSave and SaveStateNecessary are set to true so that all position and docking information is loaded and saved to the desktop files. Then an internal variables is assigned a newly constructed frame object which contains all the code for displaying the *Module Explorer* element of *Browse and Doc It*.

```
constructor TfrmDockableModuleExplorer.Create(AOwner: TComponent);

begin
  inherited;
  DeskSection := Name;
  AutoSave := True;
  SaveStateNecessary := True;
  FModuleExplorerFrame := TframeModuleExplorer.Create(Self);
  FModuleExplorerFrame.Parent := Self;
  FModuleExplorerFrame.Align := alClient;
end;
```

This method frees the form from memory first releasing the frame object.

```
destructor TfrmDockableModuleExplorer.Destroy;

begin
  FModuleExplorerFrame.Free;
  SaveStateNecessary := True;
  inherited;
end;
```

This method is called by external modules in *Browse and Doc It* to focus the *Module Explorer* if it exists and is visible.

```
procedure TfrmDockableModuleExplorer.Focus;

begin
  If FModuleExplorerFrame <> Nil Then
```

```
    If FModuleExplorerFrame.Visible Then
      If FModuleExplorerFrame.Explorer.Visible Then
        FModuleExplorerFrame.Explorer.SetFocus;
end;
```

This method creates an instance of the dockable *Module Explorer* form is the form does not already exist and is called in the main `InitialiseWizard` method for the expert. It is also called from `ShowDockableModuleExplorer`.

```
class procedure TfrmDockableModuleExplorer.CreateDockableModuleExplorer;

begin
  If Not Assigned(FormInstance) Then
    CreateDockableForm(FormInstance, TfrmDockableModuleExplorer);
end;
```

This method frees the form from memory and is called from the `Finalization` section of the main wizard module and also from the `Destructor` for the main wizard interface.

```
class procedure TfrmDockableModuleExplorer.RemoveDockableModuleExplorer;

begin
  FreeDockableForm(FormInstance);
end;
```

This method displays the dockable *Module Explorer* and is called from menu click item from *Browse and Doc It* menu.

```
class procedure TfrmDockableModuleExplorer.ShowDockableModuleExplorer;

begin
  CreateDockableModuleExplorer;
  ShowDockableForm(FormInstance);
end;
```

This method is called by a background thread once the module is parsed so that the structured view of the module can be rendered in the *Module Explorer*.

```
class procedure TfrmDockableModuleExplorer.RenderDocumentTree(
  BaseLanguageModule: TBaseLanguageModule);

begin
  If Assigned(FormInstance) Then
    If FormInstance.Visible Then
      FormInstance.FModuleExplorerFrame.RenderModule(BaseLanguageModule);
end;
```

This method hooks event handlers in the other modules to events exposed by the Module Explorer frame so that these events can be reacted to.

```
class procedure TfrmDockableModuleExplorer.HookEventHandlers(
  SelectionChangeProc: TSelectionChange; Focus, ScopeChange : TNotifyEvent);

begin
  If Assigned(FormInstance) Then
    Begin
      FormInstance.FModuleExplorerFrame.OnSelectionChange := SelectionChangeProc;
      FormInstance.FModuleExplorerFrame.OnFocus := Focus;
      FormInstance.FModuleExplorerFrame.OnRefresh := ScopeChange;
    End;
end;
```

32.2 Creating and Managing the Dockable Form

The main wizard constructor hooks the event handlers for the dockable form (note that the form has already been created by this point as it's called in the `IntialiseWizard` method – see below).

```
Constructor TBrowseAndDocItWizard.Create;

Var
  mmiMainMenu: TMainMenu;

Begin
  Inherited Create;
  TfrmDockableModuleExplorer.HookEventHandlers(SelectionChange, Focus,
    OptionsChange);
  ...
End;
```

The main wizard destructor removes the dockable form.

```
Destructor TBrowseAndDocItWizard.Destroy;

Begin
  ...
  TfrmDockableModuleExplorer.RemoveDockableModuleExplorer;
  Inherited Destroy;
End;
```

The below method is associated with a menu item for displaying the Module Explorer and it calls the `ShowDockableModuleExplorer` method.

```
Procedure TBrowseAndDocItWizard.ModuleExplorerClick(Sender: TObject);

Begin
  TfrmDockableModuleExplorer.ShowDockableModuleExplorer;
End;
```

This below method is called by a background thread to render the contents of the passed module.

```
procedure TEditorNotifier.RenderDocument(Module: TBaseLanguageModule);

begin
  TfrmDockableModuleExplorer.RenderDocumentTree(Module);
end;
```

The below method is where the dockable form is created so that it's available as soon as the wizard starts to load.

```
Function InitialiseWizard(WizardType : TWizardType) : TBrowseAndDocItWizard;

Var
  Svcs: IOTAServices;

Begin
  ...
  TfrmDockableModuleExplorer.CreateDockableModuleExplorer;
  ...
End;
```

Finally (no pun intended) the dockable form is removed in the `Finalization` section of the main wizard module.

```
Finalization
  ...
  TfrmDockableModuleExplorer.RemoveDockableModuleExplorer
End.
```

I hope this helps. It looks overly complicated and could be simplified a little but it does work and ensures that its work with both DLL and BPLs as they load and unload differently.

The code for this can be found attached to this PDF as `OTABrowseAndDocIt.zip`.

33. Syntax Highlighers

This chapter is new to this book.

The first thing you need to do when creating highlighters or any kind of parsing code is to get hold of the language grammar is in my case define the language grammar. The below text is the grammar (in Baukas-Naur form) for a text file script which is part of my Eidolon application.

```
<Goal> ::= ( <TextTable> | <DBTable> | <TimeLocationTable> | <OutputTable> |
      <RequirementsTable> )*;

<TextTable> ::= <DefinitionName> '=' 'CLASS' '(' 'TEXTTABLE' ')' <LineEnd>
                '{' <LineEnd>
                [ <TextTableDef> ]
                <FieldDef>+
                '}' <LineEnd>;

<DBTable> ::= <DefinitionName> '=' 'CLASS' '(' 'DBTABLE' ')' <LineEnd>
              '{' <LineEnd>
              [ <PrimaryConnection> ]
              <FieldDef>+
              '}' <LineEnd>;

<TimeLocationTable> ::= <DefinitionName> '=' 'CLASS' '(' 'TIMELOCATIONTABLE' ')'
      <LineEnd>
                      '{' <LineEnd>
                      <FieldDef>+
                      <TimeLocationDef>+
                      '}' <LineEnd>;

<OutputTable> ::= <DefinitionName> '=' 'CLASS' '(' 'OUTPUTTABLE' ')' <LineEnd>
                  '{' <LineEnd>
                  [ <PrimaryConnection> ]
                  [ <SecondaryConnection> ]
                  <FieldDef>+
                  '}' <LineEnd>;

<RequirementsTable> ::= <DefinitionName> '=' 'CLASS' '(' 'REQUIREMENTSTABLE' ')'
      <LineEnd>
                      '{' <LineEnd>
                      [ <PrimaryConnection> ]
                      <FieldDef>+
                      <AssociationDef>+
                      <FieldDef>+
                      <AssociationDef>+
                      '}' <LineEnd>;

<PrimaryConnection> ::= <DatabaseDef> <ConnectionDef> <TableNameDef>;

<SecondaryConnection> ::= <PrimaryConnection>;

<TextTableDef> ::= '#TABLENAME' '=' <FileName> <LineEnd>;

<DatabaseDef> ::= '#DATABASE' '=' ( <FileName> | <Directory> ) <LineEnd>;

<ConnectionDef> ::= '#CONNECTION' '=' <ConnectionString> <LineEnd>;

<TableNameDef> ::= '#TABLENAME' '=' ( <TableName> | <FileName> ) <LineEnd>;

<TTypeInfo> ::= 'B' | 'Y' | 'I' | 'L' | 'U' | 'S' | 'F' | 'D' | 'C' '('
      <ColumnWidth> ')' | 'O' | 'M';

<FieldDef> ::= [ '*' ] <InputName> ':' <TTypeInfo> [ '=' <OutputName> ] <LineEnd>;

<AssociationDef> ::= '@' <InputName> '=' <OutputName>;

<TimeLocationDef> ::= '&' <SymbolName> '=' ( <Rectangle> | <Bar> | <Line> |
      <Triangle> | <Ellipse> | <Diamond> | <SuperBar> );
```

```
<Rectangle> ::= 'RECTANGLE' ',' <BorderDef> ',' <InteriorDef> [ ',' <Transparency> ]
      <LineEnd>;

<Bar> ::= 'BAR' ',' <BorderDef> ',' <InteriorDef> ',' <BarWidth> [ ','
      <Transparency> ] <LineEnd>;

<Line> ::= 'LINE' ',' <BorderDef> [ <LineEndDefs> ] <LineEnd>;

<LineEndDefs> ::= <StartType> ',' <StartSize> ',' <EndType> ',' <EndSize>;

<StartType> ::= <LineEndType>;

<EndType> ::= <LineEndType>;

<StartSize> ::= <LineEndSize>;

<EndSize> ::= <LineEndSize>;

<LineEndType> ::= 'NONE' | 'DIAMOND' | 'OPEN' | 'OVAL' | 'STEALTH' | 'STYLEMIXED' |
      'TRIANGLE';

<LineEndSize> ::= 'ShortNarrow' | 'MediumNarrow' | 'LongNarrow' | 'ShortMedium' |
      'MediumMedium' | 'LongMedium' | 'ShortWide'   | 'MediumWide'   | 'LongWide';

<Triangle> ::= 'TRIANGLE' ',' <BorderDef> ',' <InteriorDef> ',' <TriangleType> [ ','
      <Transparency> ] <LineEnd>;

<Ellipse> ::= 'ELLIPSE' ',' <BorderDef> ',' <InteriorDef> ',' <EllipseRadius> [ ','
      <Transparency> ] <LineEnd>;

<Diamond> ::= 'DIAMOND' ',' <BorderDef> ',' <InteriorDef> ',' <DiamondSize> [ ','
      <Transparency> ] <LineEnd>;

<SuperBar> ::= 'SUPERBAR' ',' <BorderDef> ',' <InteriorDef> ',' <DateWidth> ','
      <LocationWidth> [ ',' <Transparency> ] <LineEnd>;

<BorderDef> ::= <BorderColour> ',' <BorderLineStyle> ',' <BorderWeight>;

<InteriorDef> ::= <InteriorColour> ',' <InteriorPattern> ','
      <InteriorPatternColour>;

<BorderColour> ::= <ColourName>;

<BarWidth> ::= <PositiveSingle>;

<DateWidth> ::= <PositiveSingle>;

<LocationWidth> ::= <PositiveSingle>;

<TriangleType> ::= 'STARTANDEARLY' | 'STARTANDLATE' | 'ENDANDEARLY' | 'ENDANDLATE';

<EllipseRadius> ::= <PositiveSingle>;

<DiamondSize> ::= <PositiveSingle>;

<Transparency> ::= [ ',' <PositiveSingle> ];

<ColourName> ::= 'BLACK' | 'BROWN' | 'OLIVEGREEN' | 'DARKGREEN' | 'DARKTEAL' |
      'DARKBLUE' | 'INDIGO' | 'GRAY-80%' | 'DARKRED' | 'ORANGE' | 'DARKYELLOW' |
      'GREEN' | 'TEAL' | 'BLUE' | 'BLUE-GRAY' | 'GRAY-50%' | 'RED' | 'LIGHTORANGE' |
      'LIME' | 'SEAGREEN' | 'AQUA' | 'LIGHTBLUE' | 'VIOLET' | 'GRAY-40%' | 'PINK' |
      'GOLD' | 'YELLOW' | 'BRIGHTGREEN' | 'TURQUOISE' | 'SKYBLUE' | 'PLUM' | 'GRAY-
      25%' | 'ROSE' | 'TAN' | 'LIGHTYELLOW' | 'LIGHTGREEN' | 'LIGHTTURQUOISE' |
      'PALEBLUE' | 'LAVENDER' | 'WHITE';

<BorderLineStyle> ::= 'SOLID' | 'ROUNDDOT' | 'SQUAREDOT' | 'DASH' | 'DASHDOT' |
      'LONGDASH' | 'LONGDASHDOT' | 'DASHDOTDOT';
```

```
<BorderWeight> ::= '0.25' | '0.5' | '1' | '1.5' | '2.25' | '3' | '4.5' | '6' |
      'DOUBLE' | 'DOUBLETHINTHICK' | 'DOUBLETHICKTHIN' | 'TRIPLETHICKBETWEENTHIN';

<InteriorColour> ::= <ColourName>;

<InteriorPattern> ::= 'NONE' | '10PERCENT' | '20PERCENT' | '25PERCENT' | '30PERCENT'
      | '40PERCENT' | '50PERCENT' | '5PERCENT' | '60PERCENT' | '70PERCENT' |
      '75PERCENT' | '80PERCENT' | '90PERCENT' | 'DARKDOWNWARDDIAGONAL' |
      'DARKHORIZONTAL' | 'DARKUPWARDDIAGONAL' | 'DARKVERTICAL' |
      'DASHEDDOWNWARDDIAGONAL' | 'DASHEDHORIZONTAL' | 'DASHEDUPWARDDIAGONAL' |
      'DASHEDVERTICAL' | 'DIAGONALBRICK' | 'DIVOT' | 'DOTTEDGRID' |
      'HORIZONTALBRICK' | 'LARGECHECKERBOARD' | 'LARGECONFETTI' | 'LARGEGRID' |
      'LIGHTDOWNWARDDIAGONAL' | 'LIGHTHORIZONTAL' | 'LIGHTUPWARDDIAGONAL' |
      'LIGHTVERTICAL' | 'NARROWHORIZONTAL' | 'NARROWVERTICAL' | 'OUTLINEDDIAMOND' |
      'PLAID' | 'SHINGLE' | 'SMALLCHECKERBOARD' | 'SMALLCONFETTI' | 'SMALLGRID' |
      'SOLIDDIAMOND' | 'SPHERE' | 'TRELLIS' | 'WAVE' | 'WEAVE' |
      'WIDEDOWNWARDDIAGONAL' | 'WIDEUPWARDDIAGONAL' | 'ZIGZAG';

<InteriorPatternColour> ::= <ColourName>;

<DefinitionName> ::= <Name>;

<TableName> ::= <Name>;

<ColumnWidth> ::= <ByteNumber>;

<InputName> ::= <Name>;

<OutputName> ::= <Name>;

<Name> ::= ( <Identifier> | <WhiteSpace> | <GeneralSymbols> )*;

<ByteNumber> ::= ? An integer number between 1 and 255 ?;

<PositiveSingle> ::= ? A positive single precision number ?;

<LineEnd> ::= ? Line feed and or carriage return ?;

<FileName> ::= ? Drive:\Path\FileName.Extension with no quotes ?;

<Directory> ::= ? Drive:\Path\ with no quotes ?;

<SymbolName> ::= ( <Identifier> | <WhiteSpace> )+;

<ConnectionString> ::= '' | 'dBASE IV;' | 'Paradox 4.x;' | 'FoxPro 2.6;' | 'Text;';

<AlphaCharacters> ::= ? All alphabeta characters = ['A'..'Z', 'a'..'z', '_'] ?;

<NumericCharacters> ::= ? All Numeric Characters = ['0'..'9'] ?;

<GeneralSymbols> ::= ? General Symbols = [#33..#128] - ['=', ':', '*', '(', ')',
      '#', '&', '@', '{', '}'] ?;

<WhiteSpace> ::= ? Whitespace = [#32, #9] {Space and Tab} ?;

<Identifier> ::= <AlphaCharacters> | ( <AlphaCharacters> | <NumericCharacters> )*;
```

In the below code there are two interfaces that I've implemented. The first `IOTAHighlighter` is the interface you must implement to get the code in the IDE editors to syntax highlight your code. The second interface `IOTAHighlighterPreview` provides information to the IDE's option dialogue to allow it to show you a preview of the syntax highlighting.

One of the first things to note about the first interface is that the tokenizing works on a single line of text so the methods you use must take this into account however there is another tokenizing method to help with tokens that span multiple lines, like a block comment.

Below is the definition of my highlighter for my own Eidolon MAP file information.

```
Type
```

```
  TEidolonHighlighter = Class(TNotifierObject, IOTANotifier, IOTAHighlighter {$IFDEF
      D2005}, IOTAHighlighterPreview{$ENDIF})
  {$IFDEF D2005} Strict {$ENDIF} Private
  {$IFDEF D2005} Strict {$ENDIF} Protected
  Public
    Constructor Create;
    // IOTAHighlighter methods
    Function GetIDString: String;
    Function GetName: String;
    Procedure Tokenize(StartClass: Byte; LineBuf: PAnsiChar; LineBufLen: Word;
      HighlightCodes: POTASyntaxCode);
    Function TokenizeLineClass(StartClass: Byte; LineBuf: PAnsiChar;
      LineBufLen: Word): Byte;
    // IOTAHighligherPreview methods
    Function  GetBlockEndCol: Integer;
    Function  GetBlockEndLine: Integer;
    Function  GetBlockStartCol: Integer;
    Function  GetBlockStartLine: Integer;
    Function  GetCurrentInstructionLine: Integer;
    Function  GetDisabledBreakpointLine: Integer;
    Function  GetDisplayName: String;
    Function  GetErrorLine: Integer;
    Function  GetInvalidBreakpointLine: Integer;
    Function  GetSampleSearchText: String;
    Function  GetSampleText: String;
    Function  GetValidBreakpointLine: Integer;
  End;
```

For our highlighter I've defined an alphabetical list of reserved words as below.

```
Const
  strReservedWords : Array[0..9] Of String = (
    'bar', 'class', 'dbtable', 'diamond', 'ellipse', 'line', 'rectangle',
      'texttable', 'timelocationtable', 'triangle'
    );
```

The idea originally behind the below constructor for the class was to either create the Edit Options information for the highlighter or update it if it already existed however I could not get it to work with the line which creates the new options always causing an Access Violation. I therefore abandoned this and have never got back to it to find out if it got fixed in other IDE versions.

```
Constructor TEidolonHighlighter.Create;

Var
  EditOps : IOTAEditOptions;
  iEditOps : Integer;

Begin
  EditOps := Nil;
  With (BorlandIDEServices As IOTAEditorServices) Do
    For iEditOps := 0 To EditOptionsCount - 1 Do
      If EditorOptions[iEditOps].IDString = 'Eidolon' Then
        EditOps := EditorOptions[iEditOps];
  If EditOps = Nil Then
    Begin
      // This causes an AV in the IDE - I think this is a bug in RAD Studio 2009.
      //EditOps := (BorlandIDEServices As
       IOTAEditorServices).AddEditOptions('Eidolon');
      //EditOps.Extensions := 'map';
      //EditOps.OptionsName := 'Eidolon MAP Files';
      //EditOps.SyntaxHighlighter := Self;
    End;
End;
```

33.1 IOTAHighligherPreview methods

The `GetBlockEndCol` method simply returns a column number for the end of the selected block of text in the preview.

```
Function TEidolonHighlighter.GetBlockEndCol: Integer;

Begin
  Result := 39;
End;
```

The `GetBlockEndLine` method simply returns a line number for the end of the selected block of text in the preview.

```
Function TEidolonHighlighter.GetBlockEndLine: Integer;

Begin
  Result := 12;
End;
```

The `GetBlockStartCol` method simply returns a column number for the start of the selected block of text I the preview.

```
Function TEidolonHighlighter.GetBlockStartCol: Integer;

Begin
  Result := 24;
End;
```

The `GetBlockStartLine` method simply returns a line number for the start of the selected block of text in the preview.

```
Function TEidolonHighlighter.GetBlockStartLine: Integer;

Begin
  Result := 12;
End;
```

The `GetCurrentInstructionLine` method simply returns a line number for the current instruction line. Since this doesn't apply to my MAP file format I return -1 so that it doesn't appear in the preview.

```
Function TEidolonHighlighter.GetCurrentInstructionLine: Integer;

Begin
  Result := -1;
End;
```

The `GetDisabledBreakpointLine` method simply returns a line number for line where a disabled breakpoint should be shown. Since this doesn't apply to my MAP file format I return -1 so that it doesn't appear in the preview.

```
Function TEidolonHighlighter.GetDisabledBreakpointLine: Integer;

Begin
  Result := -1;
End;
```

The `GetDisplayName` method returns the name of the highlighter preview in the editor options dialogue.

```
Function TEidolonHighlighter.GetDisplayName: String;

Begin
  Result := 'Eidolon';
End;
```

The `GetErrorLine` method simply returns a line number for line where an error should be shown. Since this doesn't apply to my MAP file format I return -1 so that it doesn't appear in the preview.

```
Function TEidolonHighlighter.GetErrorLine: Integer;
```

```
Begin
  Result := -1;
End;
```

The `GetInvalidBreakpointLine` method simply returns a line number for line where an invalid breakpoint should be shown. Since this doesn't apply to my MAP file format I return -1 so that it doesn't appear in the preview.

```
Function TEidolonHighlighter.GetInvalidBreakpointLine: Integer;

Begin
  Result := -1;
End;
```

This method should returns the text that should be highlighted as searched in the preview.

```
Function TEidolonHighlighter.GetSampleSearchText: String;

Begin
  Result := 'Date';
End;
```

This method returns the sample text that should be shown in the preview in the editor options.

```
Function TEidolonHighlighter.GetSampleText: String;

Begin
  Result :=
    '/**'#13#10 +
    ''#13#10 +
    '  Eidolon Map File'#13#10 +
    ''#13#10 +
    '**/'#13#10 +
    'This is a text file definition=Class(TextTable)'#13#10 +
    '{'#13#10 +
    '  #TableName=D:\Path\Text table.txt'#13#10 +
    '  Activity ID:C(255)'#13#10 +
    '  Activity Name:C(255)=Description'#13#10 +
    '  Start Date:D'#13#10 +
    '  Finish Date:D'#13#10 +
    '  Start Chainage:I'#13#10 +
    '  Start Chainage:I'#13#10 +
    '  Time Location Symbol:C(255)'#13#10 +
    '}'#13#10;
End;
```

The `GetValidBreakpointLine` method simply returns a line number for line where a valid breakpoint should be shown. Since this doesn't apply to my MAP file format I return -1 so that it doesn't appear in the preview.

```
Function TEidolonHighlighter.GetValidBreakpointLine: Integer;

Begin
  Result := -1;
End;
```

33.2 IOTAHighligher methods

The `GetIDString` is a unique ID string for this interface.

```
Function TEidolonHighlighter.GetIDString: String;

Begin
  Result := 'DGH.Eidolon Highlighter';
End;
```

This method returns the name of the highlighter.

```
Function TEidolonHighlighter.GetName: String;

Begin
  Result := 'Eidolon MAP Files';
End;
```

The below `Tokenize` method is the main method where the highlighting is achieved. The method provides a buffer of the editor line's character in the `LineBuf` parameter which is of a length `LineBufLen` and these characters are of type `PAnsiChar`. To highlight the line of the editor the `HighlightCodes` parameter needs to be filled with various highlighter attribute codes corresponding to the characters required highlighting. The attribute codes are as follows:

- `atWhiteSpace = 0;`
- `atComment = 1;`
- `atReservedWord = 2;`
- `atIdentifier = 3;`
- `atSymbol = 4;`
- `atString = 5;`
- `atNumber = 6;`
- `atFloat = 7;`
- `atOctal = 8; // not used in Pascal tokenizer`
- `atHex = 9;`
- `atCharacter = 10; // not used in Pascal tokenizer`
- `atPreproc = 11;`
- `atIllegal = 12; // not used in Pascal tokenizer`
- `atAssembler = 13;`
- `SyntaxOff = 14;`

So now to describe how I've coded the tokenizer. The first things I've done is to define a few types and constants to help with the parsing. `TBlockType` is an enumerate to describe the different token types that I will use in the grammar I've defined. The below I've defined a number of string / character constants which contain strings of characters that denote things like numbers, symbols, characters, etc. Then at the start of the procedure I initialise a number of variables to initial values. I use the `strToken` variable to collect the portions of the tokens as I go along – this is needed to collect a whole word to check for it being a reserved word in the grammar. I then initialise the highlighter code buffer to `$E` (`14` or `SyntaxOff`) to indicate no highlighter so that I only need to set the particular characters as I parse the line of text. Finally I loop through the character buffer (from `0` to `LineBufLen – 1`) checking each character against my grammar and updating my variables and more importantly setting the highlighter codes as I go. You will notice that for the reserved words I cannot determine whether it's a reserved word until I reach the end. So I have to retrospectively highlight the code once its determined that it's a reserved word. Lastly you should notice from the code that the `StartClass` parameter carries over the last token type from the previous line (set in the `TokenizeLineClass` method) so that you can handle tokens like block comments.

```
Procedure TEidolonHighlighter.Tokenize(StartClass: Byte; LineBuf: PAnsiChar;
  LineBufLen: Word; HighlightCodes: POTASyntaxCode);

Type
  TBlockType = (btNone, btIdentifier, btSingleLiteral, btDoubleLiteral,
     btTextDefinition, btLineComment, btBlockComment);

Const
  strAllSymbols = ([#33..#255]);
  strChars = (['a'..'z', 'A'..'Z', '-', '%']);
  strNumbers = (['0'..'9']);
  strSymbols = (strAllSymbols - strChars - strNumbers);
```

```
Var
  Codes : PAnsiChar;
  i : Integer;
  CurChar, LastCHar : AnsiChar;
  BlockType : TBlockType;
  iBlockStart : Integer;
  strToken: String;
  j: Integer;
  iToken: Integer;

Begin
  CurChar := #0;
  SetLength(strToken, 100);
  iToken := 1;
  BlockType := btNone;
  iBlockStart := 0;
  Codes := PAnsiChar(HighlightCodes);
  FillChar(HighlightCodes^, LineBufLen, $E); // No highlighter
  For i := 0 To LineBufLen - 1 Do
    Begin
      If StartClass <> atComment Then
        Begin
          LastChar := CurChar;
          CurChar := LineBuf[i];
          If ((LastChar In ['*']) And (CurChar In ['/']) And (BlockType In
      [btBlockComment])) Then
            Begin
              Codes[i] := AnsiChar(atComment);
              //BlockType := btNone;
              //iBlockStart := 0;
              Break;
            End Else
          If ((LastChar In ['/']) And (CurChar In ['/'])) Or (BlockType In
      [btLineComment]) Then
            Begin
              Codes[i - 1] := AnsiChar(atComment);
              Codes[i] := AnsiChar(atComment);
              BlockType := btLineComment;
              If iBlockStart = 0 Then
                iBlockStart := i - 1;
            End Else
          If ((LastChar In ['/']) And (CurChar In ['*'])) Or (BlockType In
      [btBlockComment]) Then
            Begin
              Codes[i - 1] := AnsiChar(atComment);
              Codes[i] := AnsiChar(atComment);
              BlockType := btBlockComment;
              If iBlockStart = 0 Then
                iBlockStart := i - 1;
            End Else
          If CurChar In strChars Then
            Begin
              Codes[i] := AnsiChar(atIdentifier);
              strToken[iToken] := Char(curChar);
              Inc(iToken);
            End
          Else If CurChar In strNumbers Then
            Codes[i] := AnsiChar(atNumber)
          Else If CurChar In strSymbols Then
            Codes[i] := AnsiChar(atSymbol);
          If (i > 0) And (Codes[i] <> AnsiChar(atIdentifier)) And
            (Codes[i - 1] = AnsiChar(atIdentifier)) Then
            Begin
              SetLength(strToken, iToken - 1);
              If IsKeyWord(strToken, strReservedWords) Then
                Begin
                  For j := i - 1 DownTo i - Length(strToken) Do
                    Codes[j] := AnsiChar(atReservedWord);
```

```
              End;
            SetLength(strToken, 100);
            iToken := 1;
          End;
      End Else
        Codes[i] := Char(atComment);
  End;
End;
```

The `TokenizeLineClass` method allows us to tell the highlighter what the last character of the previous line is so that tokens which span more than one line can be handled (like a block comment). The `Result` is set to the current `StartClass` and then modified as before if comment terminators are found in the line of text.

```
Function TEidolonHighlighter.TokenizeLineClass(StartClass: Byte; LineBuf: PAnsiChar;
  LineBufLen: Word): Byte;

Var
  i : Integer;
  LastChar, CurChar: AnsiChar;

Begin
  Result := StartClass;
  CurChar := #0;
  For i := 0 To LineBufLen - 1 Do
    Begin
      LastChar := CurChar;
      CurChar := LineBuf[i];
      If (LastChar In ['/']) And (CurChar In ['*']) Then
        Result := atComment
      Else If (LastChar In ['*']) And (CurChar In ['/']) Then
        Result := atWhiteSpace;
    End;
End;
```

33.3 Expert Creation

Finally we need to define the code that creates the highlighter and manages its lifetime.

```
Var
  ...
  iEidolonHighlighter : Integer = iWizardFailState;
  ...
```

Below the highlighter is created in the `InitaliseWizard` method where all the other elements of the package are created.

```
Function InitialiseWizard(WizardType : TWizardType) : TBrowseAndDocItWizard;

Var
  Svcs: IOTAServices;

Begin
  ...
  iEidolonHighlighter := (BorlandIDEServices As
      IOTAHighlightServices).AddHighlighter(
    TEidolonHighlighter.Create);
  ...
End;
```

Finally it's freed in the `Finalization` section of the main wizard module.

```
Finalization
  If iEidolonHighlighter > iWizardFailState Then
    (BorlandIDEServices As
      IOTAHighlightServices).RemoveHighlighter(iEidolonHighlighter);
  ...
```

```
End.
```

I hope this helps you to understand how to create a highlighter in the IDE.

The code for this can be found attached to this PDF as `OTABrowseAndDocIt.zip`.

34. Project Manager Menus

This chapter is new to this book.

The last token (for this book at the moment) is to talk about providing context menus in the Project Manager windows so that your code can provide helpful features. I have implemented this in my *Integrated Testing Helper* expert so that you can set aspects of the testing framework for each project context.

Looking back at the code below it would seem that this was not possible until the more modern Galileo IDE was introduced (2005). Also I do remember that with Delphi 2010 Borland / CodeGear made a step change in the way that this should be coded so you will see in a number of cases two completely different implementations. If you read the comments the new interface is design to work with multiple-selections where-as the original did not and is marked as deprecated.

Below is the definition of the Delphi 2005 to 2009 class for handling Project Manager menus.

```
{$IFDEF D2005}
Type
  (** A class to handle the creation of a menu for the project manager. **)
  TProjectManagerMenu = Class(TNotifierObject,
    {$IFNDEF D2010} INTAProjectMenuCreatorNotifier {$ELSE}
      IOTAProjectMenuItemCreatorNotifier {$ENDIF})
    {$IFDEF D2005} Strict {$ENDIF} Private
    FWizard: TTestingHelperWizard;
    {$IFDEF D2005} Strict {$ENDIF} Protected
    Procedure OptionsClick(Sender: TObject);
  Public
    Constructor Create(Wizard: TTestingHelperWizard);
    // INTAProjectMenuCreatorNotifier Methods
    {$IFNDEF D2010}
    Function AddMenu(Const Ident: String): TMenuItem;
    {$ELSE}
    Procedure AddMenu(Const Project: IOTAProject; Const IdentList: TStrings;
      Const ProjectManagerMenuList: IInterfaceList; IsMultiSelect: Boolean);
    {$ENDIF}
    Function CanHandle(Const Ident: String): Boolean;
    // IOTANoifier Methods
    Procedure AfterSave;
    Procedure BeforeSave;
    Procedure Destroyed;
    Procedure Modified;
  End;
```

Below is the definition of the Delphi 2010 and above class for handling Project Manager menus.

```
{$IFDEF D2010}
  (** A class to define a Delphi 2010 Project Menu Item. **)
  TITHelperProjectMenu = Class(TNotifierObject, IOTALocalMenu,
      IOTAProjectManagerMenu)
    {$IFDEF D2005} Strict {$ENDIF} Private
    FWizard  : TTestingHelperWizard;
    FProject : IOTAProject;
    FPosition: Integer;
    FCaption : String;
    FName    : String;
    FVerb    : String;
    FParent  : String;
    FSetting : TSetting;
    {$IFDEF D2005} Strict {$ENDIF} Protected
  Public
    // IOTALocalMenu Methods
    Function GetCaption: String;
    Function GetChecked: Boolean;
    Function GetEnabled: Boolean;
    Function GetHelpContext: Integer;
```

```
    Function GetName: String;
    Function GetParent: String;
    Function GetPosition: Integer;
    Function GetVerb: String;
    Procedure SetCaption(Const Value: String);
    Procedure SetChecked(Value: Boolean);
    Procedure SetEnabled(Value: Boolean);
    Procedure SetHelpContext(Value: Integer);
    Procedure SetName(Const Value: String);
    Procedure SetParent(Const Value: String);
    Procedure SetPosition(Value: Integer);
    Procedure SetVerb(Const Value: String);
    // IOTAProjectManagerMenu Methods
    Function GetIsMultiSelectable: Boolean;
    Procedure SetIsMultiSelectable(Value: Boolean);
    Procedure Execute(Const MenuContextList: IInterfaceList); Overload;
    Function PreExecute(Const MenuContextList: IInterfaceList): Boolean;
    Function PostExecute(Const MenuContextList: IInterfaceList): Boolean;
    // Constructor
    Constructor Create(Wizard: TTestingHelperWizard; Project: IOTAProject;
      strCaption, strName, strVerb, strParent: String; iPosition: Integer;
      Setting: TSetting);
  End;
{$ENDIF}
{$ENDIF}
```

34.1 INTAProjectMenuCreatorNotifier Methods

The following methods are part of the Delphi 2005 to 2009 implementation for Project manager menus.

The below `AddMenu` method is the implementation required for Delphi 2005 to 2009 and resultant `TMenuItem` will be inserted into the project manager local menu. In this particular instance I am adding sub menus to the main menu.

```
{$IFDEF D2005}

{ TProjectManagerMenu }
{$IFNDEF D2010}
Function TProjectManagerMenu.AddMenu(Const Ident: String): TMenuItem;

Var
  SM: TMenuItem;

Begin
  Result := Nil;
  If Like(sProjectContainer, Ident) Then
    Begin
      Result         := TMenuItem.Create(Nil);
      Result.Caption := strMainCaption;
      SM             := TMenuItem.Create(Nil);
      SM.Caption     := strProjectCaption;
      SM.Name        := strProjectName;
      SM.OnClick     := OptionsClick;
      Result.Add(SM);
      SM         := TMenuItem.Create(Nil);
      SM.Caption := strBeforeCaption;
      SM.Name    := strBeforeName;
      SM.OnClick := OptionsClick;
      Result.Add(SM);
      SM         := TMenuItem.Create(Nil);
      SM.Caption := strAfterCaption;
      SM.Name    := strAfterName;
      SM.OnClick := OptionsClick;
      Result.Add(SM);
      SM         := TMenuItem.Create(Nil);
      SM.Caption := strZIPCaption;
      SM.Name    := strZIPName;
      SM.OnClick := OptionsClick;
```

```
      Result.Add(SM);
    End;
End;
```

34.2 IOTAProjectMenuItemCreatorNotifier Methods

The below `AddMenu` method is the implementation required for Delphi 2010 and above.

This method is called for each menu item you wish to add to the project manager for the given list of `idents`. Add an `IOTAProjectManagerMenu` to the `ProjectManagerMenuList`. An example of a value for `IdentList` is `sFileContainer` and the name of the file. Look in the `ToolsAPI.pas` file for other constants.

```
{$ELSE}
Procedure TProjectManagerMenu.AddMenu(Const Project: IOTAProject;
  Const IdentList: TStrings; Const ProjectManagerMenuList: IInterfaceList;
  IsMultiSelect: Boolean);

Var
  i, j     : Integer;
  iPosition: Integer;
  M        : IOTAProjectManagerMenu;

Begin
  For i := 0 To IdentList.Count - 1 Do
    If sProjectContainer = IdentList[i] Then
      Begin
        iPosition := 0;
        For j     := 0 To ProjectManagerMenuList.Count - 1 Do
          Begin
            M := ProjectManagerMenuList.Items[j] As IOTAProjectManagerMenu;
            If CompareText(M.Verb, 'Options') = 0 Then
              Begin
                iPosition := M.Position + 1;
                Break;
              End;
          End;
        ProjectManagerMenuList.Add(TITHelperProjectMenu.Create(FWizard, Project,
          strMainCaption, strMainName, strMainName, '', iPosition, seProject));
        ProjectManagerMenuList.Add(TITHelperProjectMenu.Create(FWizard, Project,
          strProjectCaption, strProjectName, strProjectName, strMainName, iPosition
      + 1,
          seProject));
        ProjectManagerMenuList.Add(TITHelperProjectMenu.Create(FWizard, Project,
          strBeforeCaption, strBeforeName, strBeforeName, strMainName, iPosition +
      2,
          seBefore));
        ProjectManagerMenuList.Add(TITHelperProjectMenu.Create(FWizard, Project,
          strAfterCaption, strAfterName, strAfterName, strMainName, iPosition + 3,
          seAfter));
        ProjectManagerMenuList.Add(TITHelperProjectMenu.Create(FWizard, Project,
          strZIPCaption, strZIPName, strZIPName, strMainName, iPosition + 4,
      seZIP));
      End;
End;
{$ENDIF}
```

This method should return `True` if you wish to install a project manager menu item for this `Ident`. In cases where the Project Manager node is a file `Ident` it will be a fully qualified file name.

```
Function TProjectManagerMenu.CanHandle(Const Ident: String): Boolean;

Begin
  Result := sProjectContainer = Ident;
End;
```

34.3 IOTANotifier Methods

The following methods are part of the Delphi 2005 to 2009 implementation for Project manager menus.

This method is called immediately after the item is successfully saved. This is not called for `IOTAWizards` and I don't think its call for menus either.

```
Procedure TProjectManagerMenu.AfterSave;

Begin
End;
```

This method is called immediately before the item is saved. This is not called for `IOTAWizard` and I don't think it's called for menus either.

```
Procedure TProjectManagerMenu.BeforeSave;

Begin
End;
```

This is the constructor for the menu item – it stores a reference to the main wizard so it can call its methods.

```
Constructor TProjectManagerMenu.Create(Wizard: TTestingHelperWizard);

Begin
  FWizard := Wizard;
End;
```

If you menu item is managing any memory then is should be freed here. Exceptions are ignored.

```
Procedure TProjectManagerMenu.Destroyed;

Begin
End;
```

This method is called when associated item is modified in some way however this is not called for `IOTAWizards` and I'm not sure it's called in this context either.

```
Procedure TProjectManagerMenu.Modified;

Begin
End;
```

This below method is a simple on click event handler for the menu items which handles the menu clicks in different ways depending on the menus name which we've stated when creating them (i.e. we open different dialogues for each menu). You will also see here that I find the project context for the menu before calling the dialogues so that the dialogues can how project specific information.

```
Procedure TProjectManagerMenu.OptionsClick(Sender: TObject);

Var
  Project : IOTAProject;
  strIdent: String;

Begin
  Project := (BorlandIDEServices As
      IOTAProjectManager).GetCurrentSelection(strIdent);
  If Sender Is TMenuItem Then
    If (Sender As TMenuItem).Name = strProjectName Then
      FWizard.ConfigureOptions(Project, seProject)
    Else If (Sender As TMenuItem).Name = strBeforeName Then
      FWizard.ConfigureOptions(Project, seBefore)
    Else If (Sender As TMenuItem).Name = strAfterName Then
      FWizard.ConfigureOptions(Project, seAfter)
```

```
    Else If (Sender As TMenuItem).Name = strZIPName Then
      FWizard.ConfigureOptions(Project, seZIP);
End;
```

34.4 IOTALocalMenu Methods

The following methods are part of the Delphi 2010 and higher implementation for Project manager menus.

The below constructor creates our implementation of the Project Manager menu and stores a number of pieces of information within the class related to the menu so that we can pass context information to the menu handler.

```
{ TITHelperProjectMenu }

{$IFDEF D2010}
Constructor TITHelperProjectMenu.Create(Wizard: TTestingHelperWizard;
  Project: IOTAProject; strCaption, strName, strVerb, strParent: String;
  iPosition: Integer; Setting: TSetting);

Begin
  FWizard   := Wizard;
  FProject  := Project;
  FPosition := iPosition;
  FCaption  := strCaption;
  FName     := strName;
  FVerb     := strVerb;
  FParent   := strParent;
  FSetting  := Setting;
End;
```

This method should return the caption for the menu item including its accelerator.

```
Function TITHelperProjectMenu.GetCaption: String;

Begin
  Result := FCaption;
End;
```

This method should return whether the menu item is checked.

```
Function TITHelperProjectMenu.GetChecked: Boolean;

Begin
  Result := False;
End;
```

This method should return whether the menu item is enabled for the selected item. Note that the other methods allow you to not even show the menu for certain contexts therefore you might not need to return false here.

```
Function TITHelperProjectMenu.GetEnabled: Boolean;

Begin
  Result := True;
End;
```

This method should return the help context integer to be used for this menu item. I'm not sure how useful this will be as your menu item would probably need its own additional help information which would not be integrated with the IDE's help however you may be able to use one of the other OTA interfaces to intercept this and redirect it to your own help file (*now there's another topic to think about*).

```
Function TITHelperProjectMenu.GetHelpContext: Integer;

```

```
Begin
  Result := 0;
End;
```

This method should return the name for this menu item. If this method returns an empty string then a name will be automatically generated by the IDE.

```
Function TITHelperProjectMenu.GetName: String;

Begin
  Result := FName;
End;
```

This method should return the parent menu for this menu item.

```
Function TITHelperProjectMenu.GetParent: String;

Begin
  Result := FParent;
End;
```

This method should return the position within the parent menu where this menu item should be positioned.

```
Function TITHelperProjectMenu.GetPosition: Integer;

Begin
  Result := FPosition;
End;
```

This comment associated with this method in the `ToolsAPI.pas` file states that it returns the verb associated with this menu item. Looking through the file I could not find a reason for this method / property so not sure what exactly it does.

```
Function TITHelperProjectMenu.GetVerb: String;

Begin
  Result := FVerb;
End;
```

The comment within the `ToolsAPI.pas` file states that this sets the Caption of the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetCaption(Const Value: String);

Begin
  // Do nothing.
End;
```

The comment within the `ToolsAPI.pas` file states that this sets the Checked state of the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetChecked(Value: Boolean);

Begin
  // Do nothing.
End;
```

The comment within the `ToolsAPI.pas` file states that this sets the Enabled state of the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetEnabled(Value: Boolean);

Begin
  // Do nothing.
End;
```

The comment within the ToolsAPI.pas file states that this sets the help context of the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetHelpContext(Value: Integer);

Begin
  // Do nothing.
End;
```

The comment within the ToolsAPI.pas file states that this sets the Name of the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetName(Const Value: String);

Begin
  // Do nothing.
End;
```

The comment within the ToolsAPI.pas file states that this sets the Parent of the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetParent(Const Value: String);

Begin
  // Do nothing.
End;
```

The comment within the ToolsAPI.pas file states that this sets the position of the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetPosition(Value: Integer);

Begin
  // Do nothing.
End;
```

The comment within the ToolsAPI.pas file states that this sets the verb associated with the menu item to the specified value however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetVerb(Const Value: String);

Begin
  // Do nothing.
End;
```

34.5 IOTAProjectManagerMenu Methods

This method determines whether the menu item will appear when there are multiple items selected. In this particular case my configuration dialogues can only work with a single project at a time so I return False.

```
Function TITHelperProjectMenu.GetIsMultiSelectable: Boolean;

```

```
Begin
  Result := False;
End;
```

This method sets the multi-select property of the class. I'm not sure where you would need to set this as you are more than likely just use the above `Get` method to read the `IsMultiSelectable` property however you could possible require then menu item to be dynamic therefore you code use this to help in that regard.

```
Procedure TITHelperProjectMenu.SetIsMultiSelectable(Value: Boolean);

Begin
  // Do nothing.
End;
```

The `Execute` is called when the menu item is selected where the `MenuContextList` is a list of `IOTAProjectMenuContext`. Each item in the list represents an item in the Project Manager that is selected. In this code below the selection of the menu is delegated to the `ConfigureOptions` method of the main wizard where the appropriate configuration dialogue is displayed based on the `FSetting` field.

```
Procedure TITHelperProjectMenu.Execute(Const MenuContextList: IInterfaceList);

Begin
  FWizard.ConfigureOptions(FProject, FSetting);
End;
```

The `PreExecute` method is called before the Execute method where the `MenuContextList` is a list of `IOTAProjectMenuContext` items. Each item in the list represents an item in the Project Manager that is selected. I don't need to do any processing here so I return `False`.

```
Function TITHelperProjectMenu.PreExecute(Const MenuContextList: IInterfaceList):
      Boolean;

Begin
  Result := False;
End;
```

The `PostExecute` method is called after the Execute method where the `MenuContextList` is a list of `IOTAProjectMenuContext` items. Each item in the list represents an item in the Project Manager that is selected. I don't need to do any processing here so I return `False`.

```
Function TITHelperProjectMenu.PostExecute(Const MenuContextList: IInterfaceList):
      Boolean;

Begin
  Result := False;
End;
{$ENDIF}
{$ENDIF}
```

The code for this can be found attached to this PDF as `OTAIntegratedTestingHelper.zip`.

35. The End… for Now

Like many things if there we time enough in the world I could keep on writing able the many aspects of the OTA that are still in the source code files which haven't been covered here however I hope that there is enough information in this book to allow the more adventurous of you to seek out the knowledge from the code. One of the reason's I've always like the Delphi Object Pascal language was its strongly typed nature along with its verbose character which in my mind always seems to make it easier to read and learn from that other languages.

I hope you enjoyed the content. If you find any errors please let me know along with any constructive criticisms over contents, style, etc.

Regards

David Hoyle.

36. Index