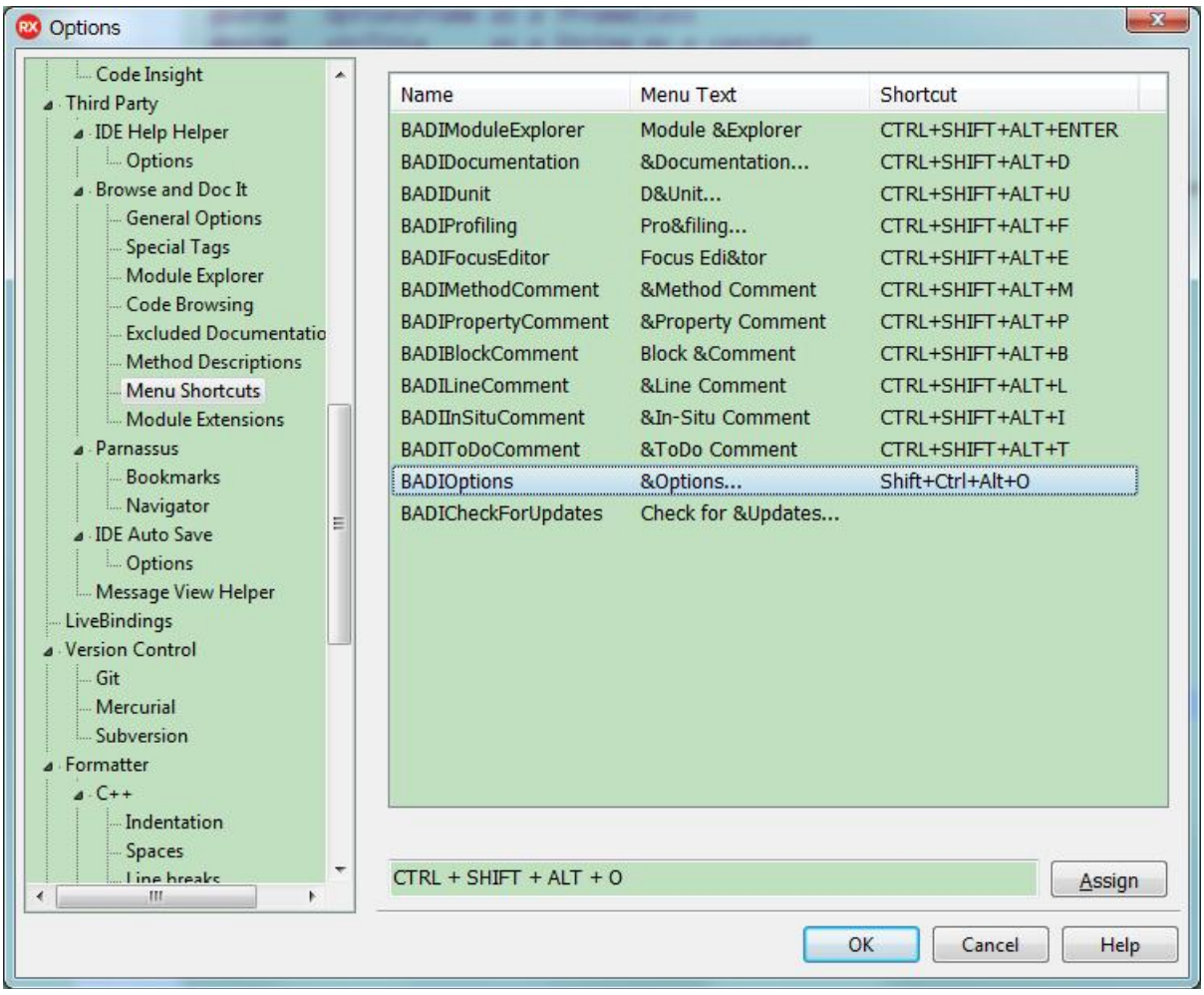
Dave's Development Blog
Software Development using Borland / Codegear / Embarcadero RAD Studio

Linking Menu Actions and Options

By David | April 9, 2017 0 Commen

Overview

In this article I want to revisit 2 aspects of the Open Tools API that I've written about before in Chapter 15: IDE Main Menus and Chapter 17: Options Page(s) inside the IDE's Options Dlg. I've been updating Browse and Doc It for a long awaited new release which updates the Object Pascal parser with the ability to handle attributes, generics and anonymous methods. Browse and Doc It hasn't had any TLC for a long time and some of the OTA code is very old (think Delphi 5 era) and so I've been updating it with some of the knowledge I've gain over the last 5 years since I've been documenting the OTA. Thus I've implemented menus that use actions which allow me to have images next to the menu items plus moving all the options information into frames for inclusion in the IDE's options dialogue.



So this article tackles a few issues:

- Fixing an issue with menu images in later IDEs;
- Providing the ability to dynamically change the menu/action shortcuts in the options dialogue;
- Decoupling the code so that the options dialogue doesn't need to know about the actions.

Implementing Menu Actions (Revisited)

I don't know about anyone else but when I look back at code I wrote 2 years ago, 5 years ago or especially 15 years ago I don't alway like what I find and want to update it in-line with what I know now. I've done this with one of my applications and its in pieces on the virtual floor and has been for over a year because I tried to do too much too quickly. So when looking back into Browse and Doc It and sometimes the articles I've written on the Open Tools API, I decide I want to change the way I do things.

In the case of action based menus in the IDE, instead of having a standalone method and storing the actions in a `TObjectList` managed in the `Intialization` and `Finalization` sections of the module, I've decided with Browse and Doc It to manage the menus and actions in a separate class. I also decided to manage the list of actions using an array of `TActions` rather than a `TObjectList` to make it easier to reference a specific action through an enumerate, hence I defined the below enumerate for the menu items.

```
  TBADIMenu = (
    bmModuleExplorer,
    bmDocumentation,
    bmDUnit,
    bmProfiling,
    bmSep1,
    bmFocusEditor,
    bmMethodComment,
    bmPropertyComment,
    bmBlockComment,
    bmLineComment,
    bmInSituComment,
    bmToDoComment,
    bmSep2,
    bmOptions,
    bmSep3,
    bmCheckForUpdates
  );
```

I also wanted to have a set of default shortcuts for the menus along with names and captions so I defined a record to hold this information as below:

```
  TBADIMenuRecord = Record
    FName      : String;
    FCaption   : String;
    FShortCut  : String;
    FMaskColor : TColor;
  End;
```

This then allowed me to defined the following constant array of default properties where the name is also used to load a bitmap from the DLL's resource file, hence the mask colour defined here.

```
  BADIMenus : Array[Low(TBADIMenu)..High(TBADIMenu)] Of TBADIMenuRecord = (
    (FName: 'BADIModuleExplorer';  FCaption: 'Module &Explorer';      FShortcut: 'CTRL+SHIFT+ALT+ENTER';
FMaskColor: clLime),
    (FName: 'BADIDocumentation';   FCaption: '&Documentation...';     FShortcut: 'CTRL+SHIFT+ALT+D';
FMaskColor: clLime),
    (FName: 'BADIDunit';           FCaption: 'D&Unit...';             FShortcut: 'CTRL+SHIFT+ALT+U';
FMaskColor: clLime),
    (FName: 'BADIProfiling';       FCaption: 'Pro&filing...';         FShortcut: 'CTRL+SHIFT+ALT+F';
FMaskColor: clLime),
    (FName: 'BADISep1';            FCaption: '';                      FShortcut: '';
FMaskColor: clLime),
    (FName: 'BADIFocusEditor';     FCaption: 'Focus Edi&tor';         FShortcut: 'CTRL+SHIFT+ALT+E';
FMaskColor: clLime),
    (FName: 'BADIMethodComment';   FCaption: '&Method Comment';       FShortcut: 'CTRL+SHIFT+ALT+M';
FMaskColor: clLime),
    (FName: 'BADIPropertyComment'; FCaption: '&Property Comment';     FShortcut: 'CTRL+SHIFT+ALT+P';
FMaskColor: clLime),
    (FName: 'BADIBlockComment';    FCaption: 'Block &Comment';        FShortcut: 'CTRL+SHIFT+ALT+B';
FMaskColor: clLime),
    (FName: 'BADILineComment';     FCaption: '&Line Comment';         FShortcut: 'CTRL+SHIFT+ALT+L';
FMaskColor: clLime),
```

```
    (FName: 'BADIInSituComment';   FCaption: '&In-Situ Comment';      FShortcut: 'CTRL+SHIFT+ALT+I';
FMaskColor: clLime),
    (FName: 'BADIToDoComment';     FCaption: '&ToDo Comment';         FShortcut: 'CTRL+SHIFT+ALT+T';
FMaskColor: clLime),
    (FName: 'BADISep2';            FCaption: '';                      FShortcut: '';
FMaskColor: clLime),
    (FName: 'BADIOptions';         FCaption: '&Options...';           FShortcut: 'CTRL+SHIFT+ALT+O';
FMaskColor: clLime),
    (FName: 'BADISep3';            FCaption: '';                      FShortcut: '';
FMaskColor: clLime),
    (FName: 'BADICheckForUpdates'; FCaption: 'Check for &Updates...'; FShortcut: '';
FMaskColor: clLime)
  );
```

With all of the above defined I could then write the constructor of my class to handle the menu / action items as follows:

```
Constructor TBADIIDEMenuInstaller.Create(Const strINIFileName : String;
  EditorNotifier : TEditorNotifier);

Var
  iImageIndex: Integer;

Begin
  NilActions;
  FINIFileName := strINIFileName;
  FEditorNotifier := EditorNotifier;
  CreateBADIMainMenu;
  iImageIndex := AddImagesToIDE;
  CreateMenuItem(FBADIMenu, bmModuleExplorer, ModuleExplorerClick, Nil, iImageIndex);
  CreateMenuItem(FBADIMenu, bmDocumentation, DocumentationClick, Nil, iImageIndex + 1);
  CreateMenuItem(FBADIMenu, bmDunit, DUnitClick, Nil, iImageIndex + 2);
  CreateMenuItem(FBADIMenu, bmProfiling, ProfilingClick, Nil, iImageIndex + 3);
  CreateMenuItem(FBADIMenu, bmSep1, Nil, Nil, 0);
  CreateMenuItem(FBADIMenu, bmFocusEditor, Focus, Nil, iImageIndex + 4);
  CreateMenuItem(FBADIMenu, bmMethodComment, MethodCommentClick, Nil, iImageIndex + 5);
  CreateMenuItem(FBADIMenu, bmPropertyComment, PropertyCommentClick, Nil, iImageIndex + 6);
  CreateMenuItem(FBADIMenu, bmBlockComment, BlockCommentClick, Nil, iImageIndex + 7);
  CreateMenuItem(FBADIMenu, bmLineComment, LineCommentClick, Nil, iImageIndex + 8);
  CreateMenuItem(FBADIMenu, bmInSituComment, InSituCommentClick, Nil, iImageIndex + 9);
  CreateMenuItem(FBADIMenu, bmToDoComment, ToDoCommentClick, Nil, iImageIndex + 10);
  CreateMenuItem(FBADIMenu, bmSep2, Nil, Nil, 0);
  CreateMenuItem(FBADIMenu, bmOptions, OptionsClick, Nil, iImageIndex + 11);
  CreateMenuItem(FBADIMenu, bmSep3, Nil, Nil, 0);
  CreateMenuItem(FBADIMenu, bmCheckForUpdates, CheckForUpdatesClick, Nil, iImageIndex + 12);
End;
```

In the above constructor there are a number of methods called to do various setup procedures. The first (and possibly redundant) method ensures that all the action references in the array are `Nil` so that I know later on which actions have been created and which haven't (separators don't have an action).

```
Procedure TBADIIDEMenuInstaller.NilActions;

Var
  iBADIMenu: TBADIMenu;

Begin
  For iBADIMenu := Low(TBADIMenu) To High(TBADIMenu) Do
    If Assigned(FBADIActions[iBADIMenu]) Then
      FBADIActions[iBADIMenu] := Nil;
End;
```

The array of actions is defined as follows:

```
  FBADIActions    : Array[Low(TBADIMenu)..High(TBADIMenu)] Of TAction;
```

Next I create the main menu item (without an action as its not required) and insert this into the IDE's main menu headings as follows:

```
Procedure TBADIIDEMenuInstaller.CreateBADIMainMenu;

Var
  mmiMainMenu: TMainMenu;

Begin
  mmiMainMenu := (BorlandIDEServices As INTAServices).MainMenu;
  FBADIMenu := TMenuItem.Create(mmiMainMenu);
  FBADIMenu.Caption := '&Browse and Doc It';
  mmiMainMenu.Items.Insert(mmiMainMenu.Items.Count - 2, FBADIMenu);
End;
```

Now for a fix to an issue I've found. When I originally wrote the article Chapter 15: IDE Main Menus on creating action based menus, I'm positive that inserting images into the IDE's image list one at a time worked as expected however I had noticed that the icons for my Integrated Testing Helper plug-in didn't look right (I had just recompiled the XE2 code) and when I started to insert images for Browse and Doc It those didn't look right either. I was getting other images against some of my menu items not the ones I wanted. So I went back to the Open Tools API code and re-read the comments and the suggestion is that you should add all your images at once and then use the returned index as the index of the first image. So the below method is all about adding the images in one go and returning the index into the IDE's image list that represents the first of your image indexes.

```
Function TBADIIDEMenuInstaller.AddImagesToIDE : Integer;

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

begin
  Result := -1;
  NTAS := (BorlandIDEServices As INTAServices);
  ilImages := TImageList.Create(Nil);
  Try
    For iMenu := Low(TBADIMenu) To High(TBADIMenu) Do
      If FindResource(hInstance, PChar(BADIMenus[iMenu].FName + 'Image'), RT_BITMAP) > 0 Then
        Begin
          BM := TBitMap.Create;
          Try
            BM.LoadFromResourceName(hInstance, BADIMenus[iMenu].FName + 'Image');
            ilImages.AddMasked(BM, BADIMenus[iMenu].FMaskColor);
          Finally
            BM.Free;
          End;
        End;
    Result := NTAS.AddImages(ilImages);
  Finally
    ilImages.Free;
  End;
end;
```

Finally with all the the above in place we can define a method to create the menu items that takes a parent menu item so that you can create nested menus, an enumerate for the specific menu to be created, on execute and update methods and finally an image index.

```
Function TBADIIDEMenuInstaller.CreateMenuItem(Const mmiParent: TMenuItem;
  Const eBADIMenu : TBADIMenu; Const ClickProc, UpdateProc : TNotifyEvent;
  iImageIndex : Integer) : TMenuItem;

Var
```

```
  NTAS: INTAServices;
  Actn : TAction;

Begin
  NTAS := (BorlandIDEServices As INTAServices);
  // Create the IDE action (cached for removal later)
  Actn := Nil;
  Result := TMenuItem.Create(NTAS.MainMenu);
  If Assigned(ClickProc) Then
    Begin
      Actn := TAction.Create(NTAS.ActionList);
      Actn.ActionList := NTAS.ActionList;
      Actn.Name := BADIMenus[eBADIMenu].FName + 'Action';
      Actn.Caption := BADIMenus[eBADIMenu].FCaption;
      Actn.OnExecute := ClickProc;
      Actn.OnUpdate := UpdateProc;
      Actn.ShortCut := TextToShortCut(TBADIOptions.BADIOptions.MenuShortcut[eBADIMenu]);
      Actn.ImageIndex := iImageIndex;
      Actn.Category := 'BADIActions';
      FBADIActions[eBADIMenu] := Actn;
    End Else
  If BADIMenus[eBADIMenu].FCaption <> '' Then
    Begin
      Result.Caption := BADIMenus[eBADIMenu].FCaption;
      Result.ImageIndex := iImageIndex;
    End Else
      Result.Caption := '-';
  // Create menu (removed through parent menu)
  Result.Action := Actn;
  Result.Name := BADIMenus[eBADIMenu].FName + 'Menu';
  // Create Action and Menu.
  mmiParent.Add(Result);
End;
```

Obviously when the plug-in is unloaded or when the IDE closes we need to clean up after ourselves so I defined the following destructor.

```
Destructor TBADIIDEMenuInstaller.Destroy;

Begin
  If FBADIMenu <> Nil Then
    FBADIMenu.Free;
  RemoveActionsFromToolbars;
  FreeActions;
  Inherited Destroy;
End;
```

Again, here I've create a number methods to do specific jobs. The `RemoveActionsFromToolbars` method is defined as before but I needed a method to free the actions that have been created as follows:

```
Procedure TBADIIDEMenuInstaller.FreeActions;

Var
  iBADIMenu: TBADIMenu;

Begin
  For iBADIMenu := Low(TBADIMenu) To High(TBADIMenu) Do
    If Assigned(FBADIActions[iBADIMenu]) Then
      FBADIActions[iBADIMenu].Free;
End;
```

All of the above creates and destroys my menu / actions but now we need to be able to edit the shortcuts in an options frame inside the IDE.

Implementing Multiple Options Frames

The second part of this article deals with the options frames and more specifically trying to use one set of code for multiple frames. I didn't quite manage a single class to handle the installation of all the frames into the IDE's option dialogue but I think you will understand why when we get there.

The first part of the puzzle it to allow the handler to load and save the frame settings to and from something. In this case I have a single global singleton object which holds all my application settings and loads and save these to and from an INI file. I've read recently that the singleton pattern is now considered an anti-pattern however without heavy refactoring, constructor injection and possibly Spring4D, I need to keep this object as is for the time being.

I could have used polymorphism to achieve the results I wanted and derive my frames from a custom frame but chose instead to create an interface (as below) that each frame needs to implement in order to be able to be loaded and saved. The reason for this approach is that at some point in the future I'll start to implemented interfaces in Browse and Doc It to decouple the code but that simple change would be far reaching to attempt now.

```
  IBADIOptionsFrame = Interface
  ['{4F8C53A5-3F4E-4B24-83F6-722F26AA8B8B}']
    Procedure LoadSettings;
    Procedure SaveSettings;
  End;
```

I've described how to implement this class before in Chapter 17: Options Page(s) inside the IDE's Options Dlg so I'll only discuss the areas of the code that have changed. With the above interface implemented by all frames then we can write the following class to handle all frames that implement this interfaces as follows:

```
TBADIIDEOptionsHandler = Class(TInterfacedObject, INTAAddInOptions)
  Strict Private
    FBADICustomFrameClass : TFrameClass;
    FBADICustomFrame      : TCustomFrame;
    FTitle                : String;
  Strict Protected
    Procedure DialogClosed(Accepted: Boolean); Virtual;
    Procedure FrameCreated(AFrame: TCustomFrame); Virtual;
    Function GetArea: String;
    Function GetCaption: String;
    Function GetFrameClass: TCustomFrameClass;
    Function GetHelpContext: Integer;
    Function IncludeInIDEInsight: Boolean;
    Function ValidateContents: Boolean;
  Public
    Constructor Create(OptionsFrame: TFrameClass; Const strTitle : String); Overload;
  End;
```

The constructor for this class simply stores the passed frame class and title string for later use in one or more of the methods.

```
Constructor TBADIIDEOptionsHandler.Create(OptionsFrame: TFrameClass; Const strTitle : String);

Begin
  FBADICustomFrameClass := OptionsFrame;
  FTitle := strTitle;
End;
```

The method `DialogClosed` has changed to check for the `IBADIOptionsFrame` interface and if present it gets a reference and calls the `SaveSettings` method of the interface.

```
Procedure TBADIIDEOptionsHandler.DialogClosed(Accepted: Boolean);

Var
  BADIOptionsFrame : IBADIOptionsFrame;

Begin
  If Accepted Then
    Begin
```

```
      If Supports(FBADICustomFrame, IBADIOptionsFrame, BADIOptionsFrame) Then
        BADIOptionsFrame.SaveSettings;
    End;
End;
```

The `FrameCreated` method is similiarly changed to check for the `IBADIOptionsFrame` interfaces and if found gets a reference and call the `LoadSettings` method of the interface.

```
Procedure TBADIIDEOptionsHandler.FrameCreated(AFrame: TCustomFrame);

Var
  BADIOptionsFrame : IBADIOptionsFrame;

Begin
  FBADICustomFrame := AFrame;
  If Supports(FBADICustomFrame, IBADIOptionsFrame, BADIOptionsFrame) Then
    BADIOptionsFrame.LoadSettings;
End;
```

Because I want each frame to have its own title under the main Browse and Doc It node in the left hand panel of the IDE's options dialogue I've altered the `GetCaption` method as below to allow me to name each frame in the constructor of this class. It also allows me to have a parent frame for the actual Browse and Doc It node with some basic version information on it (see image above).

```
Function TBADIIDEOptionsHandler.GetCaption: String;

Begin
  If FTitle <> '' Then
    Result := Format('Browse and Doc It.%s', [FTitle])
  Else
    Result := 'Browse and Doc It';
End;
```

The `GetFrameClass` method simply returns the stored frame class reference passed to the constructor.

```
Function TBADIIDEOptionsHandler.GetFrameClass: TCustomFrameClass;

Begin
  Result := FBADICustomFrameClass;
End;
```

I said at the start of this section that it was my aim to handle all the option frames using a single class as defined above however the frame for the menu / action shortcuts required the ability to trigger an update of the menu / action shortcuts and I wanted to add the ability to tell the user whether the shortcut they wanted to use was already in use. The first is required in instances where the IDE Options dialogue is invoked by the user manually and not through any menu or shortcut you have defined. The second is a nice to have and originally was implemented directly in the frame itself however when I came to compile and test the code in a standalone application I found the code would not compile because of the requirement for the `ToolsAPI.pas` unit. So this needed to be decoupled as well. I chose event handlers passed to the constructor for the simple reason that the IDE creates the frames for you so we need to hook these event handler in the options handler class. Similiarly to before I've defined an interface for one of the callbacks as follows:

```
IBADIInstallShortcutUsedCallBack = Interface
  ['{ECBC6389-DA38-4AE1-A4E9-83E6826E3776}']
    Procedure InstallShortcutUsedCallBack(ShortCutUsed : TBADIShortcutUsedEvent);
  End;
```

I created a derived class from the above options handler for this frame with a new constructor and overridden methods for `DialogClosed` and `FrameCreated`.

```
TBADIIDEShortcutOptionsHandler = Class(TBADIIDEOptionsHandler)
  Strict Private
    FUpdateEvent  : TNotifyEvent;
    FShortcutUsed : TBADIShortcutUsedEvent;
  Strict Protected
    Procedure DialogClosed(Accepted: Boolean); Override;
```

```
    Procedure FrameCreated(AFrame: TCustomFrame); Override;
  Public
    Constructor Create(OptionsFrame: TFrameClass; Const strTitle : String;
      UpdateEvent : TNotifyEvent; ShortcutUsed : TBADIShortcutUsedEvent); Overload;
  End;
```

The constuctor just stores the two event handlers for later use.

```
Constructor TBADIIDEShortcutOptionsHandler.Create(OptionsFrame: TFrameClass; Const strTitle: String;
  UpdateEvent: TNotifyEvent; ShortcutUsed : TBADIShortcutUsedEvent);

Begin
  Inherited Create(OptionsFrame, strTitle);
  FUpdateEvent := UpdateEvent;
  FShortcutUsed := ShortcutUsed;
End;
```

The `DialogClosed` method is where the update event call back is invoked to signal to the application that the menu / action shortcuts need updating.

```
Procedure TBADIIDEShortcutOptionsHandler.DialogClosed(Accepted: Boolean);

Begin
  Inherited DialogClosed(Accepted);
  If Accepted Then
    If Assigned(FUpdateEvent) Then
      FUpdateEvent(Self);
End;
```

The above invokes the below method which is defined in the Browse and Doc It Wizard class which manages all the objects in the plug-in. It calls an update method of the menu installer class.

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

Begin
  FBADIIDEMenuInstaller.UpdateMenuShortcuts;
End;
```

The menu update method is defined as below.

```
Procedure TBADIIDEMenuInstaller.UpdateMenuShortcuts;

Var
  iBADIMenu: TBADIMenu;

Begin
  For iBADIMenu := Low(TBADIMenu) To High(TBADIMenu) Do
    If Assigned(FBADIActions[iBADIMenu]) Then
      FBADIActions[iBADIMenu].ShortCut :=
        TextToShortcut(TBADIOptions.BADIOptions.MenuShortcut[iBADIMenu]);
End;
```

The `FrameCreated` method is where the call back for the checking of shortcut usage is implemented and the shortcut frame must implement the `IBADIInstallShortcutUsedCallBack` interface.

```
Procedure TBADIIDEShortcutOptionsHandler.FrameCreated(AFrame: TCustomFrame);

Var
  I : IBADIInstallShortcutUsedCallBack;

Begin
  Inherited FrameCreated(AFrame);
  If Supports(AFrame, IBADIInstallShortcutUsedCallBack, I) Then
```

```
    I.InstallShortcutUsedCallBack(FShortcutUsed);
End;
```

The above callback is implemented as follows.

```
Function TBADIIDEOptionsInstaller.IsShortcutUsed(Const iShortcut: TShortcut;
  Var strActionName : String): Boolean;

Var
  NS : INTAServices;
  iAction: Integer;

Begin
  Result := False;
  If Supports(BorlandIDEServices, INTAServices, NS) Then
    For iAction := 0 To NS.ActionList.ActionCount - 1 Do
      If NS.ActionList.Actions[iAction].ShortCut = iShortcut Then
        Begin
          strActionName := NS.ActionList.Actions[iAction].Name;
          Result := True;
        End;
End;
```

The shortcut frame can then call the callback method to check for a shortcut being in use with the below code (the proposed shortcut is in a `THotKey` control named `hkMenuShortcut`).

```
Procedure TfmBADIMenuShortcuts.hkMenuShortcutChange(Sender: TObject);

Var
  strActionName : String;

Begin
  If hkMenuShortcut.HotKey > 0 Then
    Begin
      If Assigned(FShortcutUsedEvent) And
         FShortcutUsedEvent(hkMenuShortcut.HotKey, strActionName) Then
        Begin
          lblInformation.Caption := Format('This shortcut is in use by: %s', [strActionName]);
          lblInformation.Font.Color := clRed;
          Exit;
        End;
      lblInformation.Caption := 'Shortcut not in use.';
      lblInformation.Font.Color := clGreen;
    End Else
      lblInformation.Caption := '';
End;
```

Finally we can defined another class to manage all the frame installation and removal from the IDE as follows:

```
TBADIIDEOptionsInstaller = Class
  Strict Private
    {$IFDEF DXE00}
    FBADIParentFrame      : TBADIIDEOptionsHandler;
    FBADIGeneralOptions   : TBADIIDEOptionsHandler;
    FBADISpecialtags      : TBADIIDEOptionsHandler;
    FBADIModuleExplorer   : TBADIIDEOptionsHandler;
    FBADICodeBrowsing     : TBADIIDEOptionsHandler;
    FBADIExcludedDocs     : TBADIIDEOptionsHandler;
    FBADIMethodDesc       : TBADIIDEOptionsHandler;
    FBADIMenuShortcuts    : TBADIIDEOptionsHandler;
    FBADIModuleExtensions : TBADIIDEOptionsHandler;
    {$ENDIF}
```

```
  Strict Protected
    Function IsShortcutUsed(Const iShortcut : TShortcut; Var strActionName : String) : Boolean;
  Public
    Constructor Create(UpdateMenuShortcuts : TNotifyEvent);
    Destructor Destroy; Override;
  End;
```

The constructor takes the menu update notifier event and adds all the frames to the IDE as follows:

```
Constructor TBADIIDEOptionsInstaller.Create(UpdateMenuShortcuts : TNotifyEvent);

Begin
  {$IFDEF DXE00}
  FBADIParentFrame := TBADIIDEOptionsHandler.Create(TfmBADIParentFrame, '');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADIParentFrame);
  FBADIGeneralOptions := TBADIIDEOptionsHandler.Create(TfmBADIGeneralOptions, 'General Options');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADIGeneralOptions);
  FBADISpecialtags := TBADIIDEOptionsHandler.Create(TfmBADISpecialTagsFrame, 'Special Tags');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADISpecialtags);
  FBADIModuleExplorer := TBADIIDEOptionsHandler.Create(TfmBADIModuleExplorerFrame, 'Module Explorer');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADIModuleExplorer);
  FBADICodeBrowsing := TBADIIDEOptionsHandler.Create(TfmBADICodeBrowsingFrame, 'Code Browsing');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADICodeBrowsing);
  FBADIExcludedDocs := TBADIIDEOptionsHandler.Create(TfmBADIExcludedDocFilesFrame, 'Excluded Documentation
Files');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADIExcludedDocs);
  FBADIMethodDesc := TBADIIDEOptionsHandler.Create(TfmBADIMethodDescriptionsFrame, 'Method Descriptions');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADIMethodDesc);
  FBADIMenuShortcuts := TBADIIDEShortcutOptionsHandler.Create(TfmBADIMenuShortcuts, 'Menu Shortcuts',
    UpdateMenuShortcuts, IsShortcutUsed);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADIMenuShortcuts);
  FBADIModuleExtensions := TBADIIDEOptionsHandler.Create(TfmBADIModuleExtensionsFrame, 'Module Extensions');
  (BorlandIDEServices As INTAEnvironmentOptionsServices).RegisterAddInOptions(FBADIModuleExtensions);
  {$ENDIF}
End;
```

The destructor removes the frames from the IDE as follows:

```
Destructor TBADIIDEOptionsInstaller.Destroy;

Begin
  {$IFDEF DXE00}
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADIParentFrame);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADIGeneralOptions);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADISpecialtags);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADIModuleExplorer);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADICodeBrowsing);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADIExcludedDocs);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADIMethodDesc);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADIMenuShortcuts);
  (BorlandIDEServices As INTAEnvironmentOptionsServices).UnregisterAddInOptions(FBADIModuleExtensions);
  {$ENDIF}
  Inherited Destroy;
End;
```

After Thoughts

While proofing reading this it occurred to me that there is possibly a better way to implement this by having the class that installs the menus implement an interface which has the update method as one of its method and then using dependency injection and passing this to the class that installs the options. May be that will come in the next version.

Downloads

At the moment there is no code to download but I hope to get Browse and Doc It finished before Easter (about a week away) as I have some time off between now and then.

Related posts:

1. Chapter 17: Options Page(s) inside the IDE's Options Dlg (16.9)
2. Chapter 15: IDE Main Menus (13.9)
3. Chapter 5: Useful Open Tools Utility Functions (12.1)
4. The Delphi Open Tools API Book (11)
5. Chapter 7.1: IDE Compilation Events – Revisited… (10.9)

Category: Browse and Doc It Open Tools API RAD Studio Tags: Borland, BorlandIDEServices, CodeGear, Delphi, Embarcadero, INTAAddInOptions, INTAServices, OTA, RAD Studio