Dave's Development Blog

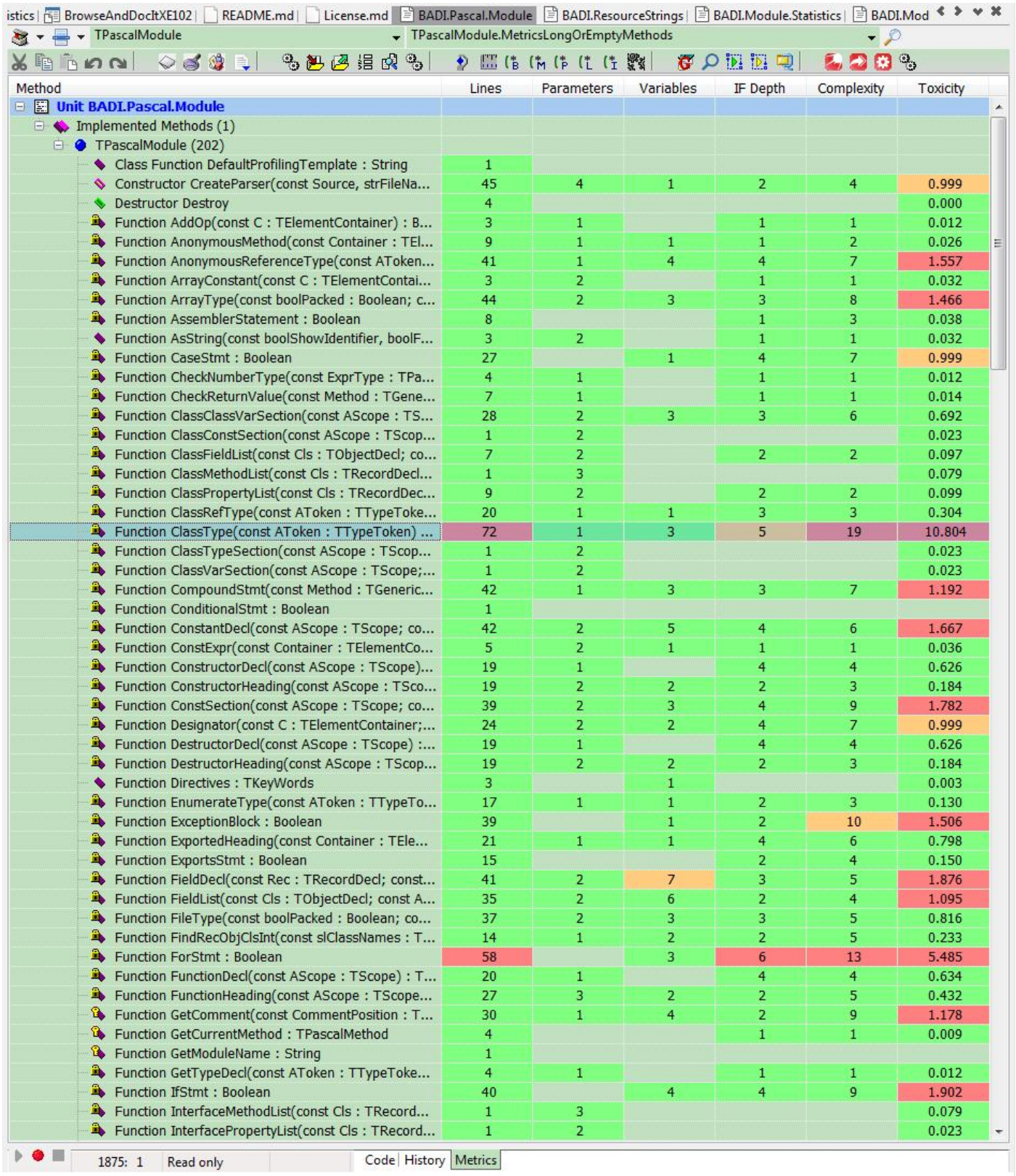
Software Development using Borland / Codegear / Embarcadero RAD Studio

RAD Studio Custom Editor Sub-views

By David | December 9, 2017 0 Commen

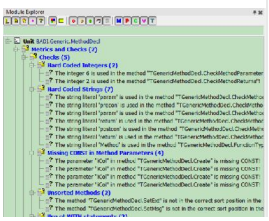
Overview

In this article I'm going to show you how to create a custom editor sub-view so that you can display information about a module in another tab in the editor (after the Code, Designer and History tabs). I was going talk about custom editor views first but I found a problem while start to write about that so I'll leave that for next time.



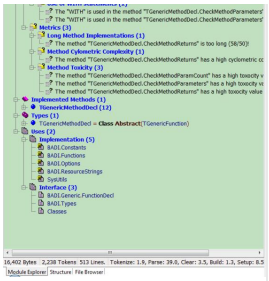
Over the course of the last month or so I've been adding support to Browse and Doc It to provide various metrics and code checks that appear in the browsing tree as you code. You may wonder why when there are products out there that can do something similar. I code for fun and not for money therefore I don't actually have a budget for any of this. I have FixInsight and its very good and I will continue to use it as it can do some static analysis my parsers cannot but I thought – How hard can it be? Not as hard as I thought, so after implementing the checks and metrics I wanted to provide an on screen report for a module (unit, project, pacage, etc) so thought I would implement a custom editor sub-view as shown below.

As you can see it shows up plenty of issues with this legancy code – that's the point in doing this.
Implementing the custom editor sub-view was quite straight-forward. So lets get cracking and show you.



Creating a Custom Editor Sub-View

The first thing to do is understand the behaviour of these sub-views in the RAD Studio IDE. The class which implements `INTACustomEditorSubView` is created only once when you register the class with the IDE and



conversely destroyed when you unregister it with the IDE. Next the IDE will create a single frame for each instance of the editor window (`TEditWindow`)) in the IDE so if you open a new window with View | New Edit WIndow then the IDE will create another instance of your frame.

You will probably think like I did that you would need to manage these frames for each edit window – **DON'T**. You will find this partically impossible to understand, at the time the frame is create, in which window it exists, as the frame is create during the construction of the edit window and therefore you cannot trace the parentage to understand it or use the `INTAEditorServices` to find the `TopEditWindow` (been there, seen it, done it and failed). You will find that for sub-views an object reference to your frame is pass to those methods where you will need it so you don't need to track frame reference.

Let's start with the definition of a class which implements the interface you will need to get a functioning custom editor sub-view (`INTACustomEditorSubView`).

Below is the declaration of the class:

```
Type
  TBADIModuleStatisticsSubView = Class(TInterfacedObject, INTACustomEditorSubView)
  Strict Private
  Strict Protected
    // INTACustomEditorSubView
    Procedure Display(Const AContext: IInterface; AViewObject: TObject);
    Function  EditAction(Const AContext: IInterface; Action: TEditAction; AViewObject: TObject): Boolean;
    Procedure FrameCreated(AFrame: TCustomFrame);
    Function  GetCanCloneView: Boolean;
    Function  GetCaption: String;
    Function  GetEditState(Const AContext: IInterface; AViewObject: TObject): TEditState;
    Function  GetFrameClass: TCustomFrameClass;
    Function  GetPriority: Integer;
    Function  GetViewIdentifier: String;
    Function  Handles(Const AContext: IInterface): Boolean;
    Procedure Hide(Const AContext: IInterface; AViewObject: TObject);
    Procedure ViewClosed(Const AContext: IInterface; AViewObject: TObject);
    // General Methods
  Public
    Class Function CreateEditorMetricsSubView : INTACustomEditorSubView;
  End;
```

You will need both `ToolsAPI` and `DesignIntf` in your uses clause.

Additionally, note that the interface is a native interfaces which means you must compile the plug-in with the same version of the IDE as the plug-in is required for (this is not necessary for IOTA interfaces). This is because the IDE is exposing its internals to you and the make up of those internals is version dependent.

INTACustomEditorSubView

This interface is the main interface to create the custom editor sub-view and has the following methods:

Procedure Display(Const AContext: IInterface; AViewObject: TObject)

This method of the interface is called when the sub-view is shown and also when its hidden (`AContext` is `Nil` when hiding). `AContext` is information provided about the module being edited and `AViewObject` is a reference to the frame object that has been created for the sub-view.

You need to use the `ContextToXxxxx` methods of the `INTAEditorServices` interface to convert a context to a specific type of information you need to process. Note that this method is called when a sub-view is being hidden and the context is nil in this instance.

```
Procedure TBADIModuleStatisticsSubView.Display(Const AContext: IInterface; AViewObject: TObject);

Var
  EVS : IOTAEditorViewServices;
  OTAModule : IOTAModule;
  Module : TBaseLanguageModule;
  strSource: String;
  SE : IOTASourceEditor;
```

```
  strEditWindowName : String;
  iIndex : Integer;

Begin
  If Supports(BorlandIDEServices, IOTAEditorViewServices, EVS) Then
    If Assigned(AContext) Then
      If EVS.ContextToModule(AContext, OTAModule) Then
        Begin
          SE := SourceEditor(OTAModule);
          strSource := EditorAsString(SE);
          Module := TBADIDispatcher.BADIDispatcher.Dispatcher(strSource, SE.FileName, SE.Modified,
            [moParse]);
          Try
            (AViewObject As TframeBADIModuleStatisticsSubView).RenderModule(Module, True);
          Finally
            Module.Free;
          End;
        End;
End;
```

In the above code I only want `IOTAModule` information so I convert the context to an `IOTAModule` reference and then extract the `IOTASourceEditor` and thus the source text which I then pass to a parser. Once parsed I cast the `AViewObject` to my custom frame to render the information as `AViewObject` is defined as the frame being displayed in the sub-view.

The `ContextToXxxxx` methods return true if the context could be converted to the given type with the reference to that type returned in the last parameter.

Function EditAction(Const AContext: IInterface; Action: TEditAction; AViewObject: TObject): Boolean

The method is invoked when an edit action from the IDE's edit menu / context menu is selected while your sub-view is active. I've implemented only Copy action here to illustrate how it works and it calls a method of my frame to copy its contents to the clipboard.

```
Function TBADIModuleStatisticsSubView.EditAction(Const AContext: IInterface; Action: TEditAction;
  AViewObject: TObject): Boolean;

Var
  strEditWindowName : String;
  iIndex : Integer;

Begin
  Result := False;
  Case Action Of
    eaCopy:
      Begin
        (AViewObject As TframeBADIModuleStatisticsSubView).CopyToClipboard;
        REsult := True;
      End;
  End;
End;
```

Again here as above I cast the `AViewObject` to my custom frame as its the sub-view being shown.

Procedure FrameCreated(AFrame: TCustomFrame)

This method of the interface is call when a frame is created by the IDE for an edit window. Here is where I originally tried to get hold of the frame reference and manage it per edit window. You don't need to do this for this interface due to the `AViewObject` being passed to various methods.

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

Begin
End;
```

Function GetCanCloneView: Boolean

This method is called to find out whether your sub-view can be cloned into a new window. Since I don't want to clone windows I have returned `False`.

```
Function TBADIModuleStatisticsSubView.GetCanCloneView: Boolean;

Begin
  Result := False;
End;
```

Function GetCaption: String

This method is called by each editor module tab to get the caption for the sub-view. I return Metrics for my sub-view tab caption.

```
Function TBADIModuleStatisticsSubView.GetCaption: String;

ResourceString
  strMetrics = 'Metrics';

Begin
  Result := strMetrics;
End;
```

Function GetEditState(Const AContext: IInterface; AViewObject: TObject): TEditState

This method is called by the IDE to get the edit capabilities of your view. You should return which edit capabilities you support in the returned set.

```
Function TBADIModuleStatisticsSubView.GetEditState(Const AContext: IInterface;
  AViewObject: TObject): TEditState;

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

Here I just support the copy action to copy the table of data in the sub-view to the clipboard.

Function GetFrameClass: TCustomFrameClass

The method is called by the IDE when it needs to create a frame for your sub-view. You should return your custom frame class reference here for the IDE to create the frame for you (do not try and create it yourself).

```
Function TBADIModuleStatisticsSubView.GetFrameClass: TCustomFrameClass;

Begin
  Result := TframeBADIModuleStatisticsSubView;
End;
```

Function GetPriority: Integer

This method is called by the IDE for each editor file so that it can position the tab control next to the existing Code, Design and History tabs.

```
Function TBADIModuleStatisticsSubView.GetPriority: Integer;

Begin
  Result := svpLow;
End;
```

The return value is an integer and the `ToolsAPI.pas` file provides some values as follows:

```
svpHighest = Low(Integer);
  svpHigh = -255;
  svpNormal = 0;
  svpLow = 255;
  svpLowest = High(Integer);
```

The Code tab is always the left-most tab. All other tabs are shown in priority order with the form designer shown at HighViewPriority. I want my sub-view to be last so I've given it a low priority.

Function GetViewIdentifier: String

This method is called to get a unique identifier for the sub-view which shouldn't be used by any other sub-views.

```
Function TBADIModuleStatisticsSubView.GetViewIdentifier: String;

Const
  strBADIMetricsSubView = 'BADICustomEditorMetricsSubView';

Begin
  Result := strBADIMetricsSubView;
End;
```

Function Handles(Const AContext: IInterface): Boolean

This method is called by the IDE before Display so that you can indicate whether or not your sub-view handles the specified context.

```
Function TBADIModuleStatisticsSubView.Handles(Const AContext: IInterface): Boolean;

Var
  EVS : IOTAEditorViewServices;
  Module: IOTAModule;

Begin
  Result := False;
  If Assigned(AContext) Then
    If Supports(BorlandIDEServices, IOTAEditorVIewServices, EVS) Then
      Result := EVS.ContextToModule(AContext, Module);
End;
```

Here, I only need to handle access to an IOTAModule interface so I test whether the context can be retrieved using one of the INTAEditorServices.ContextToXxxxx methods, specifically ContextToModule, which returns True if the context can be converted to the specified interface type.

Procedure Hide(Const AContext: IInterface; AViewObject: TObject)

This method is called by the IDE before hiding your sub-view so that you can do some processing. Display is also called at this point with a context of Nil.

```
Procedure TBADIModuleStatisticsSubView.Hide(Const AContext: IInterface; AViewObject: TObject);

Begin
  // Do nothing
End;
```

The AContext should be your custom frame for your sub-view.

Procedure ViewClosed(Const AContext: IInterface; AViewObject: TObject)

This method is called by the IDE when this interface instance is destroyed, i.e. when the edit window is closed.

```
Procedure TBADIModuleStatisticsSubView.ViewClosed(Const AContext: IInterface; AViewObject: TObject);

Begin
  // Do nothing
End;
```

You could call some clean up code here.

Class Function CreateEditorMetricsSubView : INTACustomEditorSubView

This class method is slightly superfluous as it just creates an instance of the interface (its a consequence of some of the other memory management options I was trying).

```
Class Function TBADIModuleStatisticsSubView.CreateEditorMetricsSubView: INTACustomEditorSubView;

Begin
  Result := TBADIModuleStatisticsSubView.Create;
End;
```

I just return an instance of the interface which is used in the registration of the sub-view with the IDE (see below).

Installing and Uninstalling your Custom Editor Sub-View

Now we get to the part were we can register this sub-view in the IDE. When we do regsiter the sub-view the IDE returns a pointer which we need to store for when we unregister the sub-view, so I've defined a variable in the `Implementation` section of my unit as follows:

```
Var
  ptrEditorMetricsSubView : Pointer;
```

Next we use the `INTAEditorServices` interface to register the sub-view as folows:

Installing the Sub-View

To register the sub-view we call the `RegisterEditSubView` method of the service interface `INTAEditorServices`, passing an instance of our class which implements `INTACustomEditorSubView` as follows:

```
Procedure RegisterEditorMetricsSubView;

Var
  EVS : IOTAEditorViewServices;

Begin
  If Supports(BorlandIDEServices, IOTAEditorViewServices, EVS) Then
    ptrEditorMetricsSubView := EVS.RegisterEditorSubView(
      TBADIModuleStatisticsSubView.CreateEditorMetricsSubView);
End;
```

The method is exported from the unit in which the sub-view class is defined and is called from my plug-ins wizard constructor.

Uninstalling the Sub-View

Finally we need to ensure we unregister the sub-view when the IDE closes down in a similar manner to above and using the returned pointer as follows:

```
Procedure UnregisterEditorMetricsSubView;

Var
  EVS : IOTAEditorViewServices;

Begin
  If Supports(BorlandIDEServices, IOTAEditorViewServices, EVS) Then
    EVS.UnregisterEditorSubView(ptrEditorMetricsSubView);
End;
```

The method is exported from the unit in which the sub-view class is defined and is called from my plug-ins wizard destructor.

Conclusion

As mentioned above I've also been working on custom editor views however I think these sub-views are for one, easier to implement and probably more useful for displaying information related to the module being edited. My Browse and Doc It plug-in still has a lot of work to be done to it so I cannot provide a full working example however I've attached a copy of the unit (BADI.Module.Statistics.SubView.pas) which has been referred to above.

I hope all of this has been straight forward.

Dave.

Related posts:

1. Chapter 10: Reading editor code (8.3)

2. Chapter 3: A simple custom menu (AutoSave) Fixed (7.6)
3. Chapter 8: Editor Notifiers (7.2)
4. Chapter 5: Useful Open Tools Utility Functions (7.1)
5. Chapter 2: A simple custom menu (AutoSave) (6.9)

Category: Browse and Doc It Open Tools API RAD Studio Tags: INTACustomEditorSubView, INTAEditorServices