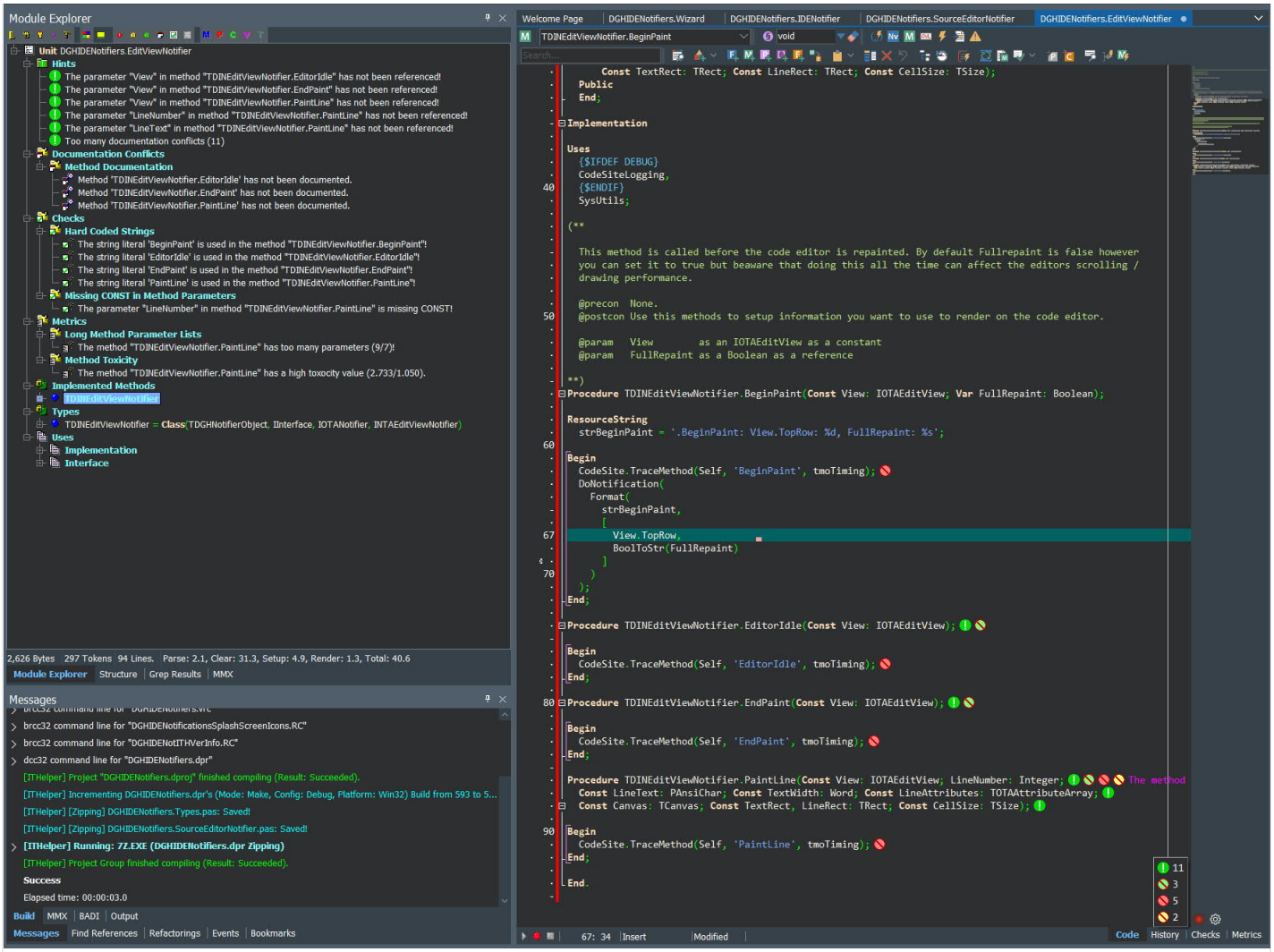
Dave's Development Blog

Software Development using Borland / Codegear / Embarcadero RAD Studio

Notify Me of Everything – Part 3…

By David | February 15, 2020 0 Comment

In this third articles on notifiers in the IDE, I'll show you how to get at some of the more difficult notifiers that aren't exposed by the `IOTAXxxxxServices` interfaces. These notifiers have been implemented in my DGH IDE Notifier plug-in but I've also used some of them in my Browse and Doc It plug-in to annotate the IDE's code editor to show where issues need to be looked at (Errors, Warning, Hints, Document Issues, Code Checks and Metrics – see the image below).

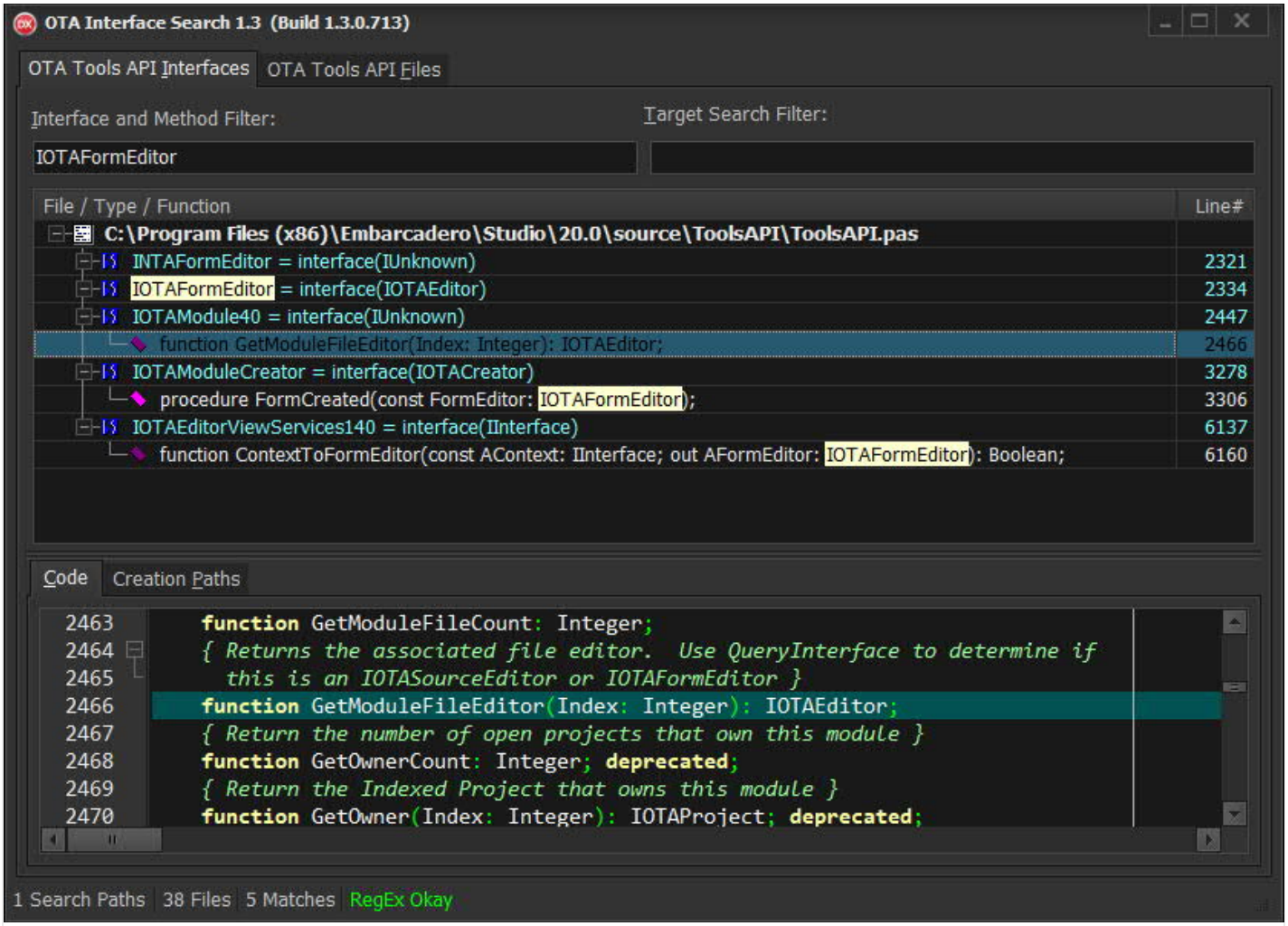


Browse and Doc It annotating the code editor

So I'll show you how to implement these and some of the issues and techniques I've used along the way.

Form and Editor Notifiers

In this section we'll look at implementing the `IOTAFormNotifier` and `IOTAEditorNotifier`s. The `IOTAFormNotifier` is exposed by the `IOTAFormEditor` interface and the `IOTAEditorNotifier` is exposed by the `IOTASourceEditor` interface. I already knew how to the to the `IOTASourceEditor` but due to some stupidity on my part, I could understand how to get to the `IOTAFormEditor` interface until I re-read the comments in the `ToolsAPI.pas` file. I use my Open Tools API Interface Search tool to find interfaces however there was a problem and that was that the information needed was only contained in a comment, so I had to updated the tool to search the comments as well.



OTA Interface Search showing the `IOTAFormEditor` reference

So now we know how to get the interfaces to install the notifiers we can create the notifiers.

IOTAFormNotifier

The `IOTAFormNotifier` definition is as below:

```
Type
  TDINFormNotifier = Class(TDNModuleNotifier, IInterface, IOTANotifier,
IOTAFormNotifier)
  Strict Private
  {$IFDEF D2010} Strict {$ENDIF} Protected
    Procedure ComponentRenamed(ComponentHandle: Pointer; Const OldName: String; Const
NewName: String);
    Procedure FormActivated;
    Procedure FormSaving;
  Public
  End;
```

The notifier interface methods are as follows:

```
Procedure TDINFormNotifier.ComponentRenamed(ComponentHandle: Pointer; Const OldName,
NewName: String);

ResourceString
   strComponentRenamed = '.ComponentRenamed = ComponentHandle: %p, OldName: %s,
NewName: %s';

Begin
  DoNotification(
    Format(
      strComponentRenamed,
      [
        ComponentHandle,
        OldName,
        NewName
      ]
    )
  );
End;
```

This method is called when a component is added or renamed on a form or data module. It passes a raw pointer to the component and the old and new component names. You can use the `IOTAFormEditor.GetComponentFromHandle` method to get a reference to the component being renamed. When a component is first created / dropped on a form this interface seems to be fired twice, the first time with both old and new names empty and the second with the old name as an empty string and the new name with the IDE's default name for the component..

```
Procedure TDINFormNotifier.FormActivated;

ResourceString
  strFormActivated = '.FormActivated';

Begin
  DoNotification(strFormActivated);
End;
```

This method is called when the form is focused in the IDE.

```
Procedure TDINFormNotifier.FormSaving;

ResourceString
   strFormSaving = '.FormSaving';

Begin
  DoNotification(strFormSaving);
```

```
End;
```

This method is called immediately prior to the form being saved (streamed to the `.DFM` file).

IOTAEditorNotifier

The definition of this notifier is as below:

```
Type
  TDINSourceEditorNotifier = Class(TDGHNotifierObject, IInterface, IOTANotifier,
IOTAEditorNotifier)
  Strict Private
    FView: IOTAEditView;
    ...
  Strict Protected
    Procedure ViewActivated(Const View: IOTAEditView);
    Procedure ViewNotification(Const View: IOTAEditView; Operation: TOperation);
  Public
    Constructor Create(Const strNotifier, strFileName : String; Const iNotification :
TDGHIDENotification; Const SourceEditor : IOTASourceEditor); ReIntroduce; Overload;
    Destructor Destroy; Override;
  End;
```

The implementation of the methods is as follows:

```
Constructor TDINSourceEditorNotifier.Create(Const strNotifier, strFileName: String;
   Const iNotification: TDGHIDENotification; Const SourceEditor : IOTASourceEditor);

Begin
  Inherited Create(strNotifier, strFileName, iNotification);
  ...
  FView := Nil;
  // Workaround for new modules create after the IDE has started
  If SourceEditor.EditViewCount > 0 Then
    ViewNotification(SourceEditor.EditViews[0], opInsert);
End;
```

For this notifier I've created a custom constructor so that I can pass to the notifier a reference to the source editor. This is required so that I can workaround a bug I found in the IDE. The `ViewNotification` method of the notifier is not always called and I found that if I created a new unit using the component palette, then I wouldn't get a `ViewNotification`. However I found that the source editor did have a view available so in this circumstance, I manually call the `ViewNotification` method. Ordinarily, when a module is opened in the IDE the source editor does not have any views available and you have to wait for the notifier to be called.

```
Destructor TDINSourceEditorNotifier.Destroy;

Begin
```

```
  ViewNotification(FView, opRemove);
  Inherited Destroy;
End;
```

The destructor above is also required to workaround another bug in the IDE where the `ViewNotification` method is not call when a view is destroyed.

```
Procedure TDINSourceEditorNotifier.ViewActivated(Const View: IOTAEditView);

ResourceString
  strViewActivate = '.ViewActivate = View.TopRow: %d';

Begin
  DoNotification(
    Format(
      strViewActivate,
      [View.TopRow]
    )
  );
End;
```

The above method is called when a view is activated / focused.

```
Procedure TDINSourceEditorNotifier.ViewNotification(Const View: IOTAEditView;
Operation: TOperation);

ResourceString
  strViewActivate = '.ViewActivate = View.TopRow: %d, Operation: %s';

Const
  strINTAEditViewNotifier = 'INTAEditViewNotifier';

Begin
  DoNotification(
    Format(
      strViewActivate,
      [
        View.TopRow,
        GetEnumName(TypeInfo(TOperation), Ord(Operation))
      ]
    )
  );
  ...
End;
```

The above method is supposed to be called when ever a view is created or destroyed however you can see

from the above that I've had to workaround a number of issues.

We will revisit some of the code above when we come to implement another notifier used for drawing on the code editor.

Implementing the Form and Editor Notifiers

First thing I've done is refactor the code to make is cleaner so the `FileNotification` method has been revised to the following:

```
Procedure TDGHNotificationsIDENotifier.FileNotification(NotifyCode:
TOTAFileNotification; Const FileName: String; Var Cancel: Boolean);

Const
  strNotifyCode : Array[Low(TOTAFileNotification)..High(TOTAFileNotification)] Of
String = (
    'ofnFileOpening',
    'ofnFileOpened',
    'ofnFileClosing',
    'ofnDefaultDesktopLoad',
    'ofnDefaultDesktopSave',
    'ofnProjectDesktopLoad',
    'ofnProjectDesktopSave',
    'ofnPackageInstalled',
    'ofnPackageUninstalled',
    'ofnActiveProjectChanged' {$IFDEF DXE80},
    'ofnProjectOpenedFromTemplate' {$ENDIF}
  );

ResourceString
  strFileNotificationNotify = '.FileNotification = NotifyCode: %s, FileName: %s,
Cancel: %s';

Var
  MS : IOTAModuleServices;
  M : IOTAModule;
  P : IOTAProject;

Begin
  DoNotification(
    Format(
    strFileNotificationNotify,
      [
        strNotifyCode[NotifyCode],
        ExtractFileName(FileName),
        strBoolean[Cancel]
```

```
      ])
  );
  If Not Cancel And Supports(BorlandIDEServices, IOTAModuleServices, MS) Then
    Case NotifyCode Of
      ofnFileOpened:
        Begin
          M := MS.OpenModule(FileName);
          If Supports(M, IOTAProject, P) Then
            Begin
              InstallProjectNotifier(M, FileName);
              InstallProjectCompileNotifier(P, FileName);
            End Else
            Begin
              InstallModuleNotifier(M, FileName);
            End;
        End;
      ofnFileClosing:
        Begin
          M := MS.OpenModule(FileName);
          If Supports(M, IOTAProject, P) Then
            Begin
              UninstallProjectNotifier(M, Filename);
              UninstallProjectCompileNotifier(P, Filename);
            End Else
            Begin
              UninstallModuleNotifier(M, Filename);
            End;
        End;
    End;
End;
```

The above allows us to them create the 2 new notifiers at the time we create the module notifier as follows:

```
Procedure TDGHNotificationsIDENotifier.InstallModuleNotifier(Const M: IOTAModule; Const
FileName: String);

Const
  strIOTAModuleNotifier = 'IOTAModuleNotifier';

Var
  MN: IOTAModuleNotifier;

Begin
  MN := TDNModuleNotifier.Create(
```

```
    strIOTAModuleNotifier,
    FileName,
    dinModuleNotifier,
    RenameModule
  );
  FModuleNotifiers.Add(FileName, M.AddNotifier(MN));
  InstallEditorNotifiers(M);
End;
```

The new method for installing the form and editor notifiers is as follows:

```
Procedure TDGHNotificationsIDENotifier.InstallEditorNotifiers(Const M: IOTAModule);

Const
  strIOTAEditViewNotifier = 'IOTAEditViewNotifier';
  strIOTAFormNotifier = 'IOTAFormNotifier';

Var
  i : Integer;
  E: IOTAEditor;
  SE : IOTASourceEditor;
  FE : IOTAFormEditor;

Begin
  For i := 0 To M.GetModuleFileCount - 1 Do
    Begin
      E := M.GetModuleFileEditor(i);
      If Supports(E, IOTASourceEditor, SE) Then
        FSourceEditorNotifiers.Add(M.FileName, SE.AddNotifier(
          TDINSourceEditorNotifier.Create(
            strIOTAEditViewNotifier,
            M.FileName,
            dinSourceEditorNotifier,
            SE
          )
        ));
      If Supports(E, IOTAFormEditor, FE) Then
        FFormEditorNotifiers.Add(M.FileName, FE.AddNotifier(
          TDINFormNotifier.Create(strIOTAFormNotifier, .FileName, dinFormNotifier)
        ));
    End;
End;
```

The above method iterates the files associated with the module (a module is a collection of files, for instance, a form has a `.pas` file and a `.dfm` file). If the module file editor for the individual file supports

either a form editor or a source editor then those implemented interfaces are used to create a form notifier or an editor notifier.

Similarly, the removal of the notifiers are as follows:

```
Procedure TDGHNotificationsIDENotifier.UninstallModuleNotifier(Const M: IOTAModule;
Const FileName: String);

Var
  MNL: IDINModuleNotifierList;
  iIndex: Integer;

Begin
  MNL := FModuleNotifiers;
  iIndex := MNL.Remove(FileName);
  If iIndex > -1 Then
    M.RemoveNotifier(iIndex);
  UninstallEditorNotifiers(M);
End;

Procedure TDGHNotificationsIDENotifier.UninstallEditorNotifiers(Const M: IOTAModule);

Var
  i: Integer;
  E: IOTAEditor;
  SE : IOTASourceEditor;
  FE : IOTAFormEditor;
  iIndex: Integer;

Begin
  For i := 0 To M.GetModuleFileCount - 1 Do
    Begin
      E := M.GetModuleFileEditor(i);
      If Supports(E, IOTASourceEditor, SE) Then
        Begin
          iIndex := FSourceEditorNotifiers.Remove(M.FileName);
          If iIndex > -1 Then
            SE.RemoveNotifier(iIndex);
        End;
      If Supports(E, IOTAFormEditor, FE) Then
        Begin
          iIndex := FFormEditorNotifiers.Remove(M.FileName);
          If iIndex > -1 Then
            FE.RemoveNotifier(iIndex);
        End;
```

```
    End;
End;
```

IOTAEditViewNotifier

Before looking at the implementation of this notifier and what you can do with it I want to revisit the editor notifier from above, more specifically the `ViewNotification` method where we add and remove the `IOTAEditViewNotifier`. Below is the modified method.

```
Procedure TDINSourceEditorNotifier.ViewNotification(Const View: IOTAEditView;
Operation: TOperation);

ResourceString
   strViewActivate = '.ViewActivate = View.TopRow: %d, Operation: %s';

Const
  strINTAEditViewNotifier = 'INTAEditViewNotifier';

Begin
  DoNotification(
    Format(
      strViewActivate,
      [
        View.TopRow,
        GetEnumName(TypeInfo(TOperation), Ord(Operation))
      ]
    )
  );
  {$IFDEF DXE100}
  Case Operation Of
    // Only create a notifier if one has not already been created!
    opInsert:
      If FEditViewNotifierIndex = -1 Then
        Begin
          FView := View;
          FEditViewNotifierIndex := View.AddNotifier(TDINEditViewNotifier.Create(
            strINTAEditViewNotifier, FileName, dinEditViewNotifier
          ));
        End;
    // opRemove Never gets called!
    opRemove:
      If FEditViewNotifierIndex > -1 Then
        Begin
          View.RemoveNotifier(FEditViewNotifierIndex);
          FEditViewNotifierIndex := -1;
```

```
        End;
  End;
  {$ENDIF DXE100}
End;
```

Above you will see that we use the insertion and removal of views to add and remove the notifier. There is a new field `FEditViewNotifierIndex` which is initialised to `-1` in the constructor to signify that the notifier has not been installed. Other than that, the passed `View` in the method is used to add and remove the notifier.

Now to the implementation of the notifier, and for this I'm going to use the code from Browse and Doc It rather than the DGH IDE Notifier above so I can show you what we can do with the notifier (note: this code is still experimental and under testing and only exists in the Development branch of the GitHub repository).

The notifier is defined as below:

```
Type
  TBADIEditViewNotifier = Class(TNotifierObject, INTAEditViewNotifier)
  Strict Private
    Const
      iPadding = 5;
    Class Var
      FFullRepaint : Boolean;
  Strict Private
    FPlainTextFontInfo   : TTokenFontInfo;
    FTokenFontInfo       : TTokenFontInfo;
    FLineHighlightColour : TColor;
    FIconsToRender       : TLimitTypes;
    FMsgsToRender        : TLimitTypes;
  Strict Protected
    // INTAEditViewNotifier
    Procedure BeginPaint(Const View: IOTAEditView; Var FullRepaint: Boolean);
    Procedure EditorIdle(Const View: IOTAEditView);
    Procedure EndPaint(Const View: IOTAEditView);
    Procedure PaintLine(Const View: IOTAEditView; LineNumber: Integer; Const LineText:
PAnsiChar; Const TextWidth: Word; Const LineAttributes: TOTAAttributeArray; Const
Canvas: TCanvas; Const TextRect: TRect; Const LineRect: TRect; Const CellSize: TSize);
    // General Methods
  Public
    Class Procedure ForceFullRepaint;
  End;
```

So lets look at the notifier methods first.

```
Procedure TBADIEditViewNotifier.BeginPaint(Const View: IOTAEditView; Var FullRepaint:
Boolean);
```

```
  Procedure IconsToRender(Const DocOps : TDocOptions; Const eDocOption : TDocOption;
Const eDocIssueType : TLimitType);

  Begin
    If eDocOption In DocOps Then
      Include(FIconsToRender, eDocIssueType);
  End;

  Procedure MsgsToRender(Const DocOps : TDocOptions; Const eDocOption : TDocOption;
Const eDocIssueType : TLimitType);

 Begin
   If eDocOption In DocOps Then
     Include(FMsgsToRender, eDocIssueType);
 End;

 Var
   DocOps: TDocOptions;
 Begin
   FullRepaint := FFullRepaint;
   FPlainTextFontInfo := TBADIOptions.BADIOptions.TokenFontInfo[True][ttPlainText];
   FTokenFontInfo := TBADIOptions.BADIOptions.TokenFontInfo[False]
[ttDocIssueEditorText];
   FLineHighlightColour := TBADIOptions.BADIOptions.TokenFontInfo[True]
[ttLineHighlight].FBackColour;
   If FTokenFontInfo.FBackColour = clNone Then
     FTokenFontInfo.FBackColour := FPlainTextFontInfo.FBackColour;
   If FTokenFontInfo.FForeColour = clNone Then
     FTokenFontInfo.FForeColour := FPlainTextFontInfo.FForeColour;
   DocOps := TBADIOptions.BADIOptions.Options;
   FIconsToRender := [];
   IconsToRender(DocOps, doShowErrorIconsInEditor, ltErrors);
   IconsToRender(DocOps, doShowWarningIconsInEditor, ltWarnings);
   IconsToRender(DocOps, doShowHintIconsInEditor, ltHints);
   IconsToRender(DocOps, doShowConflictIconsInEditor, ltConflicts);
   IconsToRender(DocOps, doShowCheckIconsInEditor, ltChecks);
   IconsToRender(DocOps, doShowMetricIconsInEditor, ltMetrics);
   FMsgsToRender := [];
   MsgsToRender(DocOps, doShowErrorMsgsInEditor, ltErrors);
   MsgsToRender(DocOps, doShowWarningMsgsInEditor, ltWarnings);
   MsgsToRender(DocOps, doShowHintMsgsInEditor, ltHints);
   MsgsToRender(DocOps, doShowConflictMsgsInEditor, ltConflicts);
   MsgsToRender(DocOps, doShowCheckMsgsInEditor, ltChecks);
```

```
    MsgsToRender(DocOps, doShowMetricMsgsInEditor, ltMetrics);
  End;
```

This method is called before the editor is painted. It is passed two parameters: the first the view being painted and the second a var parameter to determine whether the code editor should update just those lines that have changed or repaint the whole editor. Ideally you do not want to set FullRepaint to true as this will cause the editor to slow down as it redraws every line in the editor however there are circumstances where you may want this. For me in this instance, this is determined by a class varaible `FFullRepaint`. The rest of the code in this method is all about setting up information for the painting of the individual code editor lines.

When using this notifier you want to minimise the processing you do when painting a line so here I setup what I want to paint and store that information in the fields of the notifier.

```
Procedure TBADIEditViewNotifier.EditorIdle(Const View: IOTAEditView);

Begin
End;
```

This method is called a number of time when the code editor is not doing anything. I've not used this method but you could do some processing here but you would need to test how it inhibits the performance of the code editor.

```
Procedure TBADIEditViewNotifier.EndPaint(Const View: IOTAEditView);

Const
  strTEditControlClsName = 'TEditControl';

Var
  R : TRect;
  C: TWinControl;
  i: Integer;

Begin
  R := Rect(0, 0, 0, 0);
  C := View.GetEditWindow.Form;
  For i := 0 To C.ComponentCount - 1 Do
    If CompareText(C.Components[i].ClassName, strTEditControlClsName) = 0  Then
      Begin
        C := (C.Components[i] As TWinControl);
        R := C.ClientRect;
        R.TopLeft := C.ClientToScreen(R.TopLeft);
        R.BottomRight := C.ClientToScreen(R.BottomRight);
        Dec(R.Bottom, iPadding);
        Dec(R.Right, iPadding);
        Break;
      End;
```

```
  If FFullRepaint And Application.MainForm.Visible Then
    TBADIDocIssueHintWindow.Display(R, TfrmDockableModuleExplorer.DocIssueTotals);
  FFullRepaint := False;
End;
```

The above method is called when the code editor has been updated. Here I originally wanted to draw a small summary of the status of the code editor contains in the bottom right corner however there is no access to the editor view canvas and if you keep a reference from the last line paint you will find like I did that you cannot do anything with it. So in the above I decided to create a custom hint window to display the information and show it in the bottom right corner (I have to traverse the IDE and find the code editor to find the bottom right corner coordinates). The code also resets the class variable for full repainting to false.

```
Procedure TBADIEditViewNotifier.PaintLine(Const View: IOTAEditView; LineNumber:
Integer; Const LineText: PAnsiChar; Const TextWidth: Word; Const LineAttributes:
TOTAAttributeArray; Const Canvas: TCanvas; Const TextRect, LineRect: TRect; Const
CellSize: TSize);

  Procedure DrawMsgText(Var R : TRect; Const strText : String);

  Var
    strTextToRender : String;
    setFontStyles : TFontStyles;

  Begin
    strTextToRender := strText;
    setFontStyles := Canvas.Font.Style;
    Canvas.Font.Style := FTokenFontInfo.FStyles;
    Canvas.Font.Color := FTokenFontInfo.FForeColour;
    SetBkMode(Canvas.Handle, TRANSPARENT);
    DrawText(Canvas.Handle, PChar(strTextToRender), Length(strTextToRender), R, DT_LEFT
Or DT_VCENTER);
    Inc(R.Left, Canvas.TextWidth(strTextToRender));
    Canvas.Font.Style := setFontStyles;
  End;

Var
  R : TRect;
  LineDocIssue : IBADILineDocIssues;
  eLimitType: TLimitType;

Begin
  LineDocIssue := TfrmDockableModuleExplorer.LineDocIssue(LineNumber);
  R := LineRect;
  R.Left := TextRect.Right;
  InflateRect(R, -iPadding, 0);
```

```
  If Assigned(LineDocIssue) Then
    For eLimitType := Low(TLimitType) To High(TLimitType) Do
      If eLimitType In LineDocIssue.Issues Then
        Begin
          If eLimitType In FIconsToRender Then
            Begin
              Case eLimitType Of
                ltErrors:    DrawIcon(Canvas, R, ltErrors);
                ltWarnings:  DrawIcon(Canvas, R, ltWarnings);
                ltHints:     DrawIcon(Canvas, R, ltHints);
                ltConflicts: DrawIcon(Canvas, R, ltConflicts);
                ltChecks:    DrawIcon(Canvas, R, ltChecks);
                ltMetrics:   DrawIcon(Canvas, R, ltMetrics);
              End;
              Inc(R.Left, iPadding);
            End;
          If eLimitType In FMsgsToRender Then
            Begin
              DrawMsgText(R, LineDocIssue.Message[eLimitType]);
              Inc(R.Left, iPadding);
            End;
        End;
End;
```

Above is called for each line to be painted. You have access to the code editor canvas. The `LineRect` are the coordinates of the entire line from the left edge of the gutter to the right edge of the code editor. The `TextRect` are the coordinates of the text. Here I find the right edge of the text and draw either an icon or a text message (or both) for any issues I have. The issues are processed in another thread and stored as public properties of the module explorer from where they are retrieved when needed. The other parameters provide access to the code editor text rendered in the line and the syntax highlight attributes along with the size of a text character.

In the above is a local method for rendering text on the editor. You should notice that I use the Windows API call to render the text as I wanted to render the text with a transparent background so I didn't have to workout what the colour of the editor background is I'm rendering text over. The font rendering is the same as the editor but you can changes this BUT reset it when you've finished else some strange things will happen. The font colour / style is from the information setup in the `BeforePaint` method.

```
Class Procedure TBADIEditViewNotifier.ForceFullRepaint;

Begin
  FFullRepaint := True;
End;
```

This method is not part of the notifier but is allows me to set the class variable to full repaint. This is called then my background thread has finished parsing the code and it wants to get the code editor to update with the information from the parsed code.

```
Procedure TEditorNotifier.RenderDocument(Const Module: TBaseLanguageModule);

Var
  EditorSvcs : IOTAEditorServices;

begin
  TfrmDockableModuleExplorer.RenderDocumentTree(Module);
  If Supports(BorlandIDEServices, IOTAEditorServices, EditorSvcs) Then
    Begin
      If doFollowEditorCursor In TBADIOptions.BADIOptions.Options Then
        If Assigned(EditorSvcs.TopView) Then

TfrmDockableModuleExplorer.FollowEditorCursor(EditorSvcs.TopView.CursorPos.Line);
      If Assigned(EditorSvcs.TopView) Then
        Begin
          TBADIEditViewNotifier.ForceFullRepaint;
          EditorSvcs.TopView.Paint;
        End;
    End;
end;
```

The above method is called when the background thread has successfully completed parsing the code and you can see I force the full repaint and then ask the editors top view to paint.

Hope this provide food for thought.

regards
Dave.

Category: Browse and Doc It Delphi IDE Notifications Open Tools API RAD Studio Tags:

INTAEditLineNotifier, IOTAEditorNotifier, IOTAFormEditor, IOTAFormNotifier, IOTASourceEditor