Dave's Development Blog

Software Development using Borland / Codegear / Embarcadero RAD Studio

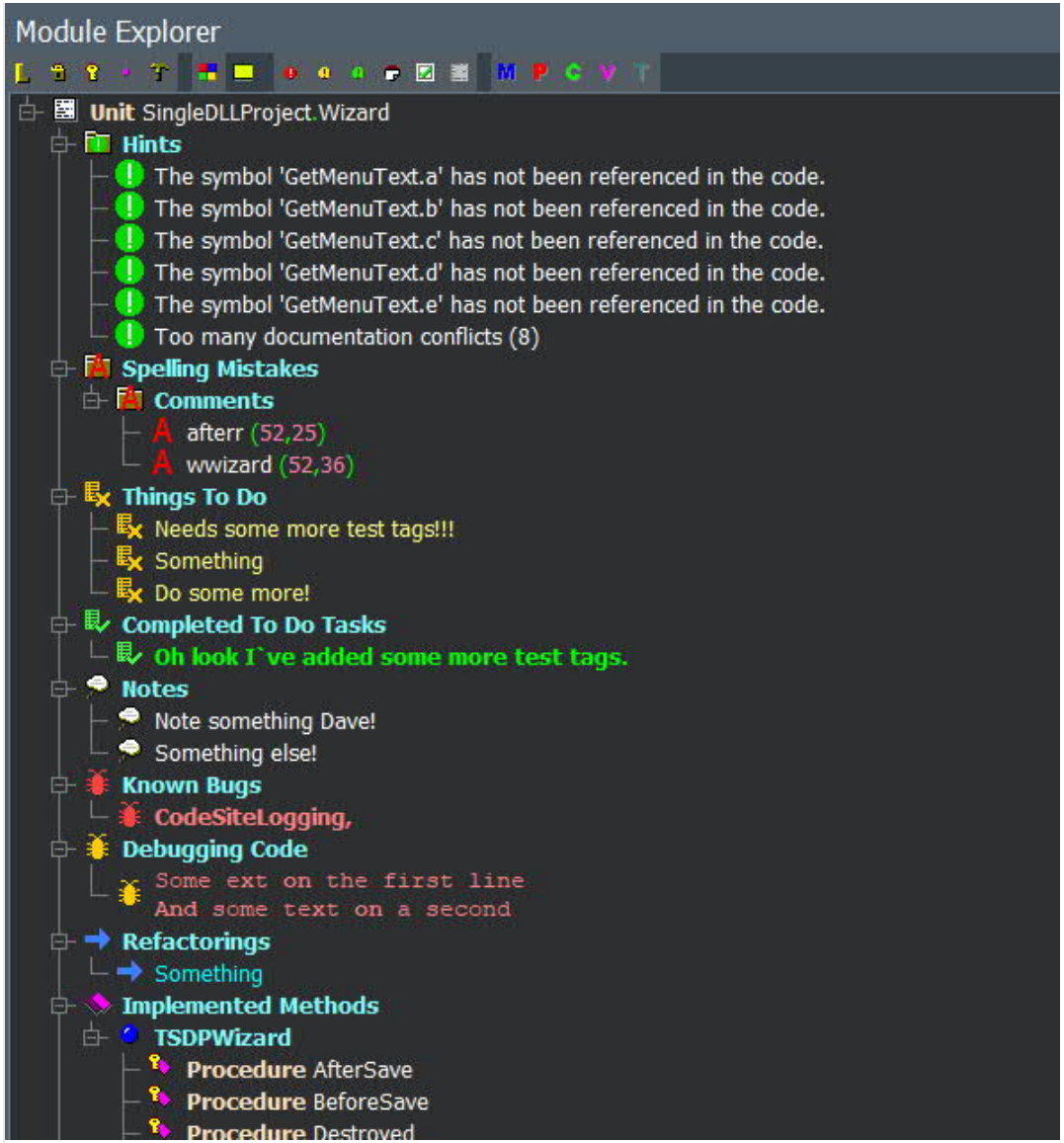
Code Editor Experiments

by David | August 8, 2020 0 Comment

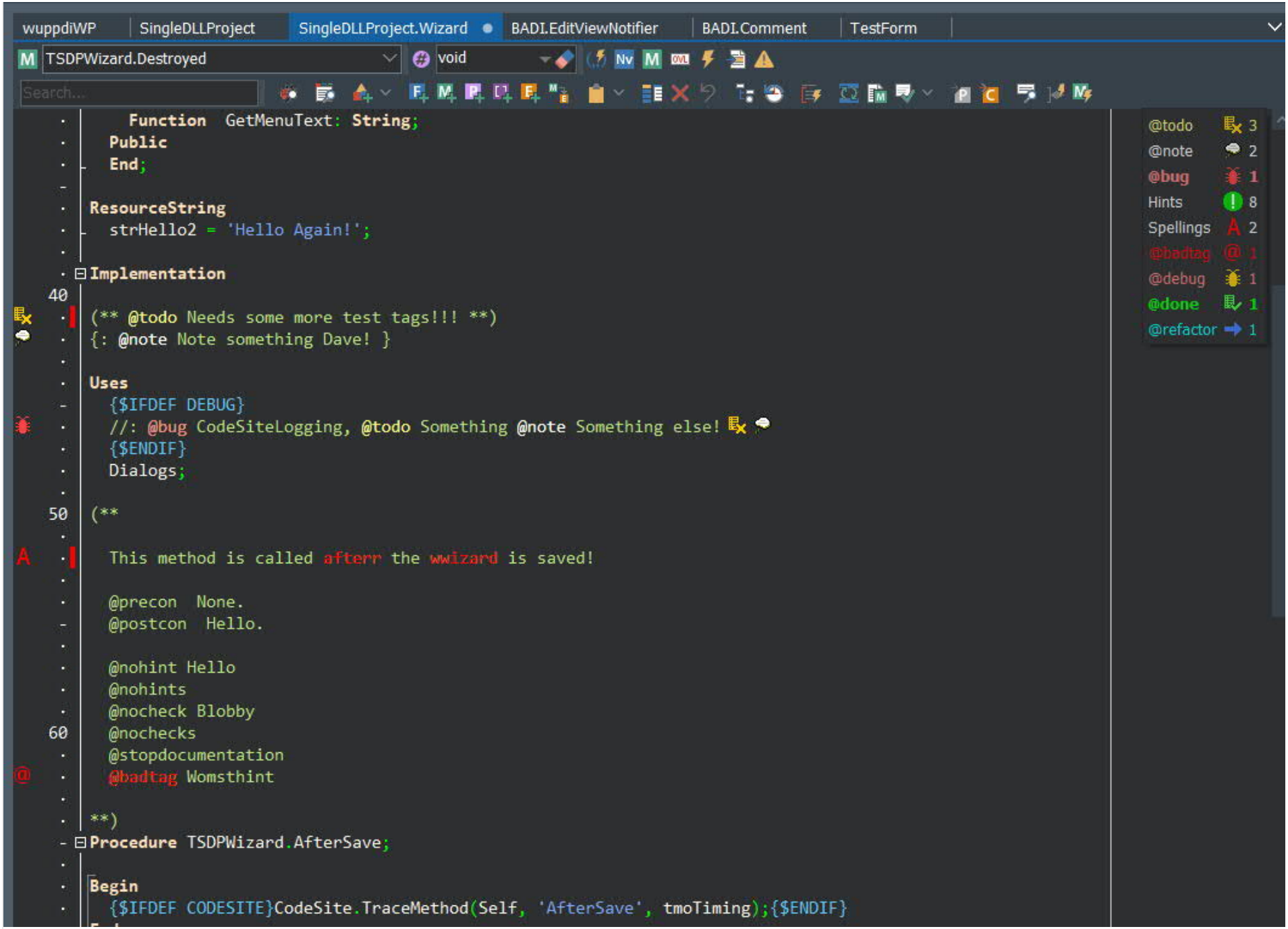
Overview

I've been trying to write this article for weeks, if not months, but kept wanting to have more a polished code base to go through, then a user of my software pointed out some spelling mistakes in my code and that prompted me to implement a spell checker for comments and string literals. Now that's done (to a point), I can now actually write about what I've done so others can perhaps use the knowledge.

So, what have I done? Well, I thought it would be good for the errors, warnings, hints, checks, metrics and documentation issues that are identified by Browse and Doc It to be annotated in the code editor. So below is some of the output from Browse and Doc It that is shown in the Module Explorer for the current code editor:



These are then shown in the code editor as icons with optional text as below:


wuppdiWP
SingleDLLProject
SingleDLLProject.Wizard
BADI.EditViewNotifier
BADI.Comment
TestForm
TSDPWizard.Destroyed
void
Search...
Function GetMenuText: String;
Public
End;
ResourceString
strHello2 = 'Hello Again!';
Implementation
(** @todo Needs some more test tags!!! **)
{: @note Note something Dave! }
Uses
{$IFDEF DEBUG}
//: @bug CodeSiteLogging, @todo Something @note Something else!
{$ENDIF}
Dialogs;
(**
This method is called afterr the wwizard is saved!
@precon None.
@postcon Hello.
@nohint Hello
@nohints
@nocheck Blobby
@nochecks
@stopdocumentation
@badtag Womsthint
**)
Procedure TSDPWizard.AfterSave;
Begin
{$IFDEF CODESITE}CodeSite.TraceMethod(Self, 'AfterSave', tmoTiming);{$ENDIF}
@todo 3
@note 2
@bug 1
Hints 8
Spellings 2
@badtag 1
@debug 1
@done 1
@refactor 1

So below, I'll go through how I've done this. It's not all pretty, for instance, the data used by the code editor is stored in the Module Explorer at the moment and really needs to be held somewhere independently.

Edit View Notifiers

It all starts with adding an `INTAEditViewNotifier` to the IDE for the current editor. I've covered this in a previous article (Notify Me of Everything – Part 3) so I won't go into details of that interface here but I will go through in detail how I've used it to achieve the above.

Notifier

Let's start with the definition of the notifier as shown below:

```
Type
  TBADIEditViewNotifier = Class(TNotifierObject, INTAEditViewNotifier)
  Strict Private
    Const
      (** A constant to define the padding between the editor content, doc issue icons and text. **)
      iPadding = 5;
    Class Var
      (** A class variable to determine whether the paint cycle is a full cycle or not. **)
      FFullRepaint : Boolean;
  Strict Private
    FPlainTextFontInfo   : TTokenFontInfo;
    FCommentFontInfo     : TTokenFontInfo;
    FTokenFontInfo       : TTokenFontInfo;
    FIconsToRender       : TStringList;
    FMsgsToRender        : TStringList;
    FHorizontalScroll    : Integer;
    FRTTIContext         : TRttiContext;
  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
    Procedure DrawMsgText(Const Canvas : TCanvas; Var R : TRect; Const strText : String;
      Const LineDocIssue : TBADIDocIssueInfo);
    Procedure DrawCommentTag(Const Canvas : TCanvas; Var R : TRect; Const strText : String;
      Const LineDocIssue : TBADIDocIssueInfo);
      procedure FindHorizontalScrollPosition(const View: IOTAEditView);
    Procedure MarkUpdateSpecialTags(Const Canvas : TCanvas;  Const strDocIssue : String;
      Const DocIssueInfo : TBADIDocIssueInfo; Const LineText: PAnsiChar; Const TextRect : TRect;
      Const CellSize: TSize);
    Procedure IconsToRender(Const DocOps : TDocOptions; Const eDocOption : TDocOption;
      Const eDocIssueType : TLimitType);
    Procedure MsgsToRender(Const DocOps : TDocOptions; Const eDocOption : TDocOption;
      Const eDocIssueType : TLimitType);
    Procedure HighlightSpellingMistakes(Const Canvas : TCanvas;
      Const LineDocIssues : IBADILineDocIssues; Const LineText: PAnsiChar; Const TextRect : TRect;
      Const CellSize: TSize);
      procedure RenderIcons(const recDocIssue: TBADIDocIssueInfo; const Canvas:
          TCanvas; const TextRect, LineRect: TRect; var R: TRect);
    Procedure RenderMsgs(Const recDocIssue: TBADIDocIssueInfo; Const Canvas: TCanvas;
      Const TextRect: TRect; Const CellSize: TSize; Var R: TRect);
  Public
    Constructor Create;
    Destructor Destroy; Override;
    Class Procedure ForceFullRepaint;
  End;
```

The above contains a number of fields and additional methods that are not part of the OTA interface. The methods will be discussed as we get to them but perhaps I should explain the fields:

Fields

`FFullRepaint : Boolean` – This is a class variable to hold a boolean value the determines whether we force a full repaint of the code editor (required as my plug-in will not necessarily have parsed the code in the editor at the time the editor wants to render the information).

`FPlainTextFontInfo : TTokenFontInfo`, `FCommentFontInfo : TTokenFontInfo` and `FTokenFontInfo : TTokenFontInfo` – These fields contain font information for rendering plain text, comments and tokens which match the fonts used in the code editor. This information is retrieved from the registry as the OTA does not provide access to this information.

`FIconsToRender : TStringList` and `FMsgsToRender : TStringList` – These are string lists of tags that the code uses to know whether to render icons and messages in the editor. Strings are used rather than enumerates as the tagging system in Browse and Doc It is dynamic so that you can define your own tags for annotating the code.

`FHorizontalScroll : Integer` – This caches the horizontal scroll position of the editor. This will be covered in more detail later on as the OTA again does not provide this information.

`FRTTIContext : TRttiContext` – This is a cached instance as we need to use this often to get the above scroll position.

Paint Cycles

Before we move on the rest of the code it's important to know the painting cycle used by the IDE as this affects the order in which the notifier interface methods are called and why I've done something in a particular order.

The rendering of the code editor starts with the `BeginPaint()` method. We use this to setup information we will need to know in order to render the annotations in the code editor. This is always called at the start of the updating of the code editor.

Then the IDE renders the lines of the code editor and calls `PaintLine()` for each line rendered. Note: the IDE only updates those lines it believes need to be updated, i.e. it does not render the whole editor each time unless you tell it to but you need to be careful here as rendering the whole editor each time will slow down the responsiveness of the code editor when typing. Therefore you need to ensure you are doing the minimum of work in this method.

Once the IDE has finished rendering the lines it calls the `EndPaint()` method to signify that painting has finished.

Finally, the `EditorIdle()` method is called periodically when the code editor is idle.

Constants

Finally, since we have to resort to hacking the IDE for some information. I've defined the below constant which is the component class name for the editor control in the IDE. More about this later.

```
Const
  (** A constant for the name of the IDE Edit Control. **)
```

```
  strTEditControlClsName = 'TEditControl';
```

Construction and Destruction

Below are the constructor and destructor for the notifier. There is nothing special about them, they just create the string lists and the RTTI context for use later on (and obviously destroys the string lists when finished).

```
Constructor TBADIEditViewNotifier.Create;

Begin
  Inherited Create;
  FIconsToRender := TStringList.Create;
  FIconsToRender.Sorted := True;
  FIconsToRender.Duplicates := dupIgnore;
  FMsgsToRender := TStringList.Create;
  FMsgsToRender.Sorted := True;
  FMsgsToRender.Duplicates := dupIgnore;
  FRTTIContext := TRTTIContext.Create;
End;

Destructor TBADIEditViewNotifier.Destroy;

Begin
  FIconsToRender.Free;
  FMsgsToRender.Free;
  Inherited Destroy;
End;
```

BeginPaint()

As described above this method is called at the start of the paint cycle. The method gives you access to the editor view and the ability to force a full repaint.

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

```
Const
  aRenderIconInfo : Array[TLimitType] Of TDocOption = (
    doShowErrorIconsInEditor,
    doShowWarningIconsInEditor,
    doShowHintIconsInEditor,
    doShowConflictIconsInEditor,
    doShowCheckIconsInEditor,
    doShowMetricIconsInEditor,
    doShowSpellingIconsInEditor
  );
  aRenderMsgInfo : Array[TLimitType] Of TDocOption = (
    doShowErrorMsgsInEditor,
    doShowWarningMsgsInEditor,
    doShowHintMsgsInEditor,
    doShowConflictMsgsInEditor,
    doShowCheckMsgsInEditor,
    doShowMetricMsgsInEditor,
    doShowSpellingMsgsInEditor
  );

Var
  DocOps: TDocOptions;
  iTag: Integer;
  eLimitType: TLimitType;

Begin
  {$IFDEF CODESITE}CodeSite.TraceMethod(Self, 'BeginPaint', tmoTiming);{$ENDIF}
  FullRepaint := FFullRepaint;
  FPlainTextFontInfo := TBADIOptions.BADIOptions.TokenFontInfo[True][ttPlainText];
  FCommentFontInfo := TBADIOptions.BADIOptions.TokenFontInfo[True][ttCommentText];
  FTokenFontInfo := TBADIOptions.BADIOptions.TokenFontInfo[False][ttDocIssueEditorText];
```

```
  If FTokenFontInfo.FBackColour = clNone Then
    FTokenFontInfo.FBackColour := FPlainTextFontInfo.FBackColour;
  If FTokenFontInfo.FForeColour = clNone Then
    FTokenFontInfo.FForeColour := FPlainTextFontInfo.FForeColour;
  DocOps := TBADIOptions.BADIOptions.Options;
  FIconsToRender.Clear;
  For eLimitType := Low(TLimitType) To High(TLimitType) Do
    IconsToRender(DocOps, aRenderIconInfo[eLimitType], eLimitType);
  For iTag := 0 To TBADIOptions.BADIOptions.SpecialTags.Count - 1 Do
    If tpShowInEditor In TBADIOptions.BADIOptions.SpecialTags[iTag].FTagProperties Then
      FIconsToRender.Add(TBADIOptions.BADIOptions.SpecialTags[iTag].FName);
  FMsgsToRender.Clear;
  For eLimitType := Low(TLimitType) To High(TLimitType) Do
    MsgsToRender(DocOps, aRenderMsgInfo[eLimitType], eLimitType);
  FindHorizontalScrollPosition(View);
End;
```

The above code does the following:

- Set's whether the code editor should be fully repainted based on the class variable (which is set elsewhere);
- Retrieves font information from the Browse and Doc It options (you can either use the IDE's colours or define your own);
- It then does some checks for background colours and patches the information accordingly;
- Caches a set of document options which determine what should be rendered in the editor;
- Using the document options, updates the string list to contain tag names for those items that should be rendered (you have quite a lot of control over what is and is not rendered);
- Finds the horizontal scroll position. We need this to determine how we render text information over the editor.

Finding the Editor's Horizontal Scroll Position

I was surprised when I looked at the `IOTAEditView` interface that I could not determine the horizontal scroll position of the editor. I searched the rest of the OTA code and could not find a workaround so had to resort to using RTTI to extract the information from the IDE.

The reason we need this is that when you are overwriting text in the editor in order to highlight it (colour comment tags and spelling mistakes) you have to adjust for the scroll position as the `PaintLine()` method does not provide the information.

Below is the code I use to get the scroll position in the above method:

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

Const
  strSHScrollPosFieldName = 'sHScrollPos';

Var
  F: TCustomForm;
  iComponent: Integer;
  Typ: TRttiType;
  Field : TRttiField;
  Value: TValue;

Begin
  FHorizontalScroll := 0;
  If Not Assigned(View) Or Not Assigned(View.GetEditWindow) Then
    Exit;
  F := View.GetEditWindow.Form;
  If Assigned(F) Then
    For iComponent := 1 To F.ComponentCount - 1 Do
      If CompareText(F.Components[iComponent].ClassName, strTEditControlClsName) = 0 Then
        Begin
          Typ := FRTTIContext.GetType(F.Components[iComponent].ClassType);
          Field := Typ.GetField(strSHScrollPosFieldName);
          If Assigned(Field) Then
            Begin
              Value := Field.GetValue(F.Components[iComponent]);
              FHorizontalScroll := Value.AsInteger;
            End;
        End;
End;
```

From the `View` parameter of `BeginPaint()` we can get the edit window form in the IDE. We can then iterate through the form's components to find the

edit control (the constant previously defined). Once we find the component, we can then find the horizontal scrollbar property and read its value. The value we get returned is the scroll position in editor columns (characters), not a pixel position.

PaintLine()

The `PaintLine()` method is where most of the work is done but first I should explain the parameters you are provided with (Note: this is called after the editor line has been rendered by the IDE):

- `View` – This provides access to the view edit being rendered;
- `LineNumber` – This is the line number in the editor that is being rendered;
- `LineText` – This is the text of the line that has been rendered in the editor;
- `TextWidth` – This is the length of the line text above;
- `LineAttributes` – This is an array of highlighter attributes that tells you the type of each character;
- `Canvas` – This provides access to the editor's canvas so you can draw on it;
- `TextRect` – This is the rectangle that encompasses the text in the editor;
- `LineRect` – This is the rectangle for the whole line from the left edge of the editor gutter to the scroll bar on the right;
- `CellSize` – This is a structure that tells you the number of pixels for the height and width of a character in the editor.

Next, I need to explain how I've thought through the annotations. The editor has room on the left in the gutter for a single icon. Note, if you render an icon here you can overwrite other icons by other IDE plug-ins and the IDE itself (blue marks for line compiled). If I have more than one icon to render for a line, I then put the next icon to the right of the editor text. Once all the icons are rendered, then the messages are rendered to the right of the editor text and icons.

Spelling mistakes and special tag highlighting are different in that the code overwrites the text in the editor to highlight the issues.

```
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);

Var
  R : TRect;
  LineDocIssues : IBADILineDocIssues;
  strDocIssue: String;
  recDocIssue : TBADIDocIssueInfo;

Begin
  {$IFDEF CODESITE}CodeSite.TraceMethod(Self, 'PaintLine', tmoTiming);{$ENDIF}
```

```
  LineDocIssues := TfrmDockableModuleExplorer.LineDocIssue(LineNumber);
  R := LineRect;
  If Assigned(LineDocIssues) Then
    For strDocIssue In LineDocIssues.Issues Do
      Begin
        recDocIssue := LineDocIssues[strDocIssue];
        RenderIcons(recDocIssue, Canvas, TextRect, LineRect, R);
        RenderMsgs(recDocIssue, Canvas, TextRect, CellSize, R);
        MarkUpdateSpecialTags(Canvas, strDocIssue, recDocIssue, LineText, TextRect, CellSize);
        If CompareText(strDocIssue, astrLimitType[ltSpelling]) = 0 Then
          HighlightSpellingMistakes(Canvas, LineDocIssues, LineText, TextRect, CellSize);
     End;
End;
```

The above code askes the Module Explorer for a LineDocIssues interfaced object for the given line. If the result is `Nil` then there are no issues with the current line being rendered. It then cycles through the issues (tags) one at a time and renders the icons and text accordingly.

Let's look at some of the methods called:

RenderIcons()

This method uses an image list in Browse and Doc It which contains all the icons that can be rendered.

```
Procedure TBADIEditViewNotifier.RenderIcons(Const recDocIssue: TBADIDocIssueInfo; Const Canvas: TCanvas;
  Const TextRect, LineRect: TRect; Var R: TRect);

Var
  iIndex: Integer;

Begin
  If FIconsToRender.Find(recDocIssue.FName, iIndex) Or (recDocIssue.FImageIndex In [iiBadTag]) Then
    Begin
      TBADIOptions.BADIOptions.ScopeImageList.Draw(
        Canvas,
```

```
        R.Left,
        R.Top,
        BADIImageIndex(recDocIssue.FImageIndex, scNone)
        );
      Inc(R.Left, TBADIOptions.BADIOptions.ScopeImageList.Width + iPadding);
      // After first icon change to Text Rect
      If (R.Left > LineRect.Left) And (R.Left < TextRect.Right) Then
        R.Left := TextRect.Right + iPadding;
    End;
End;
```

The image list is used to draw the indexed icon on the editor's canvas using the given rectangle. The lefthand side of the rectangle is moved by the width of the image plus a small margin for padding. If this consumes the icon area on the left of the editor, the rectangle is updated to the immediate right of the editor text.

RenderMsgs()

This method is similar to the previous method.

```
Procedure TBADIEditViewNotifier.RenderMsgs(Const recDocIssue: TBADIDocIssueInfo;
  Const Canvas: TCanvas; Const TextRect: TRect; Const CellSize: TSize; Var R: TRect);

Var
  iIndex: Integer;

Begin
  If FMsgsToRender.Find(recDocIssue.FName, iIndex) Then
    Begin
      If R.Left < TextRect.Right Then
        R.Left := TextRect.Right + CellSize.cx;
      DrawMsgText(Canvas, R, recDocIssue.FMessage, recDocIssue);
      Inc(R.Left, iPadding);
    End;
```

```
End;
```

It checks that the rectangle is to the right of the text then renders the text and moves the rectangle's left to the end of the rendered text

Below is the method used to render the text on the code editor.

```
Procedure TBADIEditViewNotifier.DrawMsgText(Const Canvas : TCanvas; Var R : TRect;
  Const strText : String; Const LineDocIssue : TBADIDocIssueInfo);

Var
  strTextToRender : String;
  setFontStyles : TFontStyles;

Begin
  strTextToRender := strText;
  setFontStyles := Canvas.Font.Style;
  Canvas.Font.Style := FTokenFontInfo.FStyles;
  Canvas.Font.Color := FTokenFontInfo.FForeColour;
  If LineDocIssue.FForeColour <> clNone Then
    Canvas.Font.Color := LineDocIssue.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;
```

First, it stores a copy of the current editor's font style as we need to reset this when we are finished else the editor does not always render the code editor

properly afterwards. It updates the canvas's font colour and style, patches up the font colour if none has been defined. We then use the Windows API to set a transparent background so we don't have to mess about to working out what the background colours are in the editor. Finally, the rectangle is updated and the font style reset.

MarkUpdateSpecialTags()

This method is used to overwrite the special tags to highlight them (they are part of the document comments).

```
Procedure TBADIEditViewNotifier.MarkUpdateSpecialTags(Const Canvas : TCanvas; Const strDocIssue : String;
  Const DocIssueInfo : TBADIDocIssueInfo; Const LineText: PAnsiChar; Const TextRect: TRect;
  Const CellSize: TSize);

Var
  R: TRect;
  iPos: Integer;
  strText: String;

Begin
  If strDocIssue[1] = '@' Then
    Begin
      strText := UTF8ToString(LineText);
      iPos := Pos(LowerCase(strDocIssue), LowerCase(strText));
      If iPos > 0 Then
        Begin
          strText := Copy(strText, iPos, strDocIssue.Length);
          R := TextRect;
          iPos := iPos - 1 - FHorizontalScroll;
          If iPos > 0 Then
            Inc(R.Left, iPos * CellSize.cx)
          Else
            Delete(strText, 1, -iPos);
          If strText.Length > 0 Then
            DrawCommentTag(Canvas, R, strText, DocIssueInfo);
```

```
        End;
    End;
End;
```

First, we get a copy of the text in the editor and find the position of the tag in the line (it only expects the tag to appear once in the text). If found we then get the text for the tag from the editor text. This is done so that the highlighting uses the same capitalisation as the text in the editor. We then draw the text on the editor overwriting the editor's text.

The below method does most of the work of rendering the text.

```
Procedure TBADIEditViewNotifier.DrawCommentTag(Const Canvas : TCanvas; Var R : TRect;
  Const strText : String; Const LineDocIssue : TBADIDocIssueInfo);

Var
  strTextToRender : String;
  setFontStyles : TFontStyles;

Begin
  strTextToRender := strText;
  setFontStyles := Canvas.Font.Style;
  Try
    Canvas.Brush.Color := FCommentFontInfo.FBackColour;
    If LineDocIssue.FBackColour <> clNone Then
      Canvas.Brush.Color := LineDocIssue.FBackColour
    Else
      SetBkMode(Canvas.Handle, TRANSPARENT);
    Canvas.Font.Style := FCommentFontInfo.FStyles;
    Canvas.Font.Color := FCommentFontInfo.FForeColour;
    If LineDocIssue.FForeColour <> clNone Then
      Canvas.Font.Color := LineDocIssue.FForeColour;
    DrawText(
      Canvas.Handle,
      PChar(strTextToRender),
```

```
      Length(strTextToRender),
      R,
      DT_LEFT Or DT_VCENTER
    );
  Finally
    Canvas.Font.Style := setFontStyles;
  End;
End;
```

We set up the canvas with the font properties of the comment syntax style in the editor before rendering the text. Note: we also store and reset the font style of the editor here.

HighlightSpellingMistakes()

This method is very similar to the above but is used for the highlighting of the spelling mistakes.

```
Procedure TBADIEditViewNotifier.HighlightSpellingMistakes(Const Canvas : TCanvas;
  Const LineDocIssues : IBADILineDocIssues; Const LineText: PAnsiChar; Const TextRect: TRect;
  Const CellSize: TSize);

Var
  i : Integer;
  recSpellingMistake: TBADISpellingMistake;
  R: TRect;
  iPos: Integer;
  strText: String;
  DocIssueInfo: TBADIDocIssueInfo;

Begin
  For i := 0 To LineDocIssues.SpellingMistakeCount - 1 Do
    Begin
      strText := UTF8ToString(LineText);
      recSpellingMistake := LineDocIssues.SpellingMistake[i];
```

```
      iPos := Pos(LowerCase(recSpellingMistake.FWord), LowerCase(strText), recSpellingMistake.FColumn);
      If iPos > 0 Then
        Begin
          strText := Copy(strText, iPos, recSpellingMistake.FWord.Length);
          R := TextRect;
          iPos := iPos - 1 - FHorizontalScroll;
          If iPos > 0 Then
            Inc(R.Left, iPos * CellSize.cx)
          Else
            Delete(strText, 1, -iPos);
          DocIssueInfo.FBackColour := clNone;
          DocIssueInfo.FForeColour := TBADIOptions.BADIOptions.SpellingMistakeColour;
          If strText.Length > 0 Then
            DrawCommentTag(Canvas, R, strText, DocIssueInfo);
        End;
    End;
End;
```

It finds the word that is misspelt in the editor text using the column position of the spelling mistake to ensure we highlight the correct text. It then extracts the text from the editor to get the right capitalisation and sets the fore and background colours to be used to render the text and then finally render the text with the previously referred to method.

EndPaint()

`EndPaint()` is called when the paint cycle ends.

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

Begin
  {$IFDEF CODESITE}CodeSite.TraceMethod(Self, 'EndPaint', tmoTiming);{$ENDIF}
  If FFullRepaint And Application.MainForm.Visible And (Application.MainForm.WindowState <> wsMinimized) Then
    TBADIDocIssueHintWindow.Display(TfrmDockableModuleExplorer.DocIssueTotals);
  FFullRepaint := False;
```

```
End;
```

Here we ask the document issue hint window to update (the window that shows statistics on the issues in the editor and resets the full repaint class variable to false.

EditorIdle()

In my implementation, the `EditorIdle()` method does nothing.

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

Begin
  {$IFDEF CODESITE}CodeSite.TraceMethod(Self, 'EditorIdle', tmoTiming);{$ENDIF}
End;
```

Miscellaneous Methods

The below methods are used in the `BeginPaint()` method to update the strings list.

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

Begin
  If eDocOption In DocOps Then
    FIconsToRender.Add(astrLimitType[eDocIssueType]);
End;

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

Begin
  If eDocOption In DocOps Then
    FMsgsToRender.Add(astrLimitType[eDocIssueType]);
End;
```

Conclusion

Hopefully the above helps you with your own ideas for rendering information on the code editor. Do bear in mind that whatever you render, you run the possibility of overwriting information rendered by the IDE or other IDE plug-ins and you are not provided with any information by the OTA to help in this regards.

regards

Dave.

Category: Browse and Doc It Delphi Open Tools API RAD Studio Tags: INTAEditLineNotifier, IOTAEditView