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

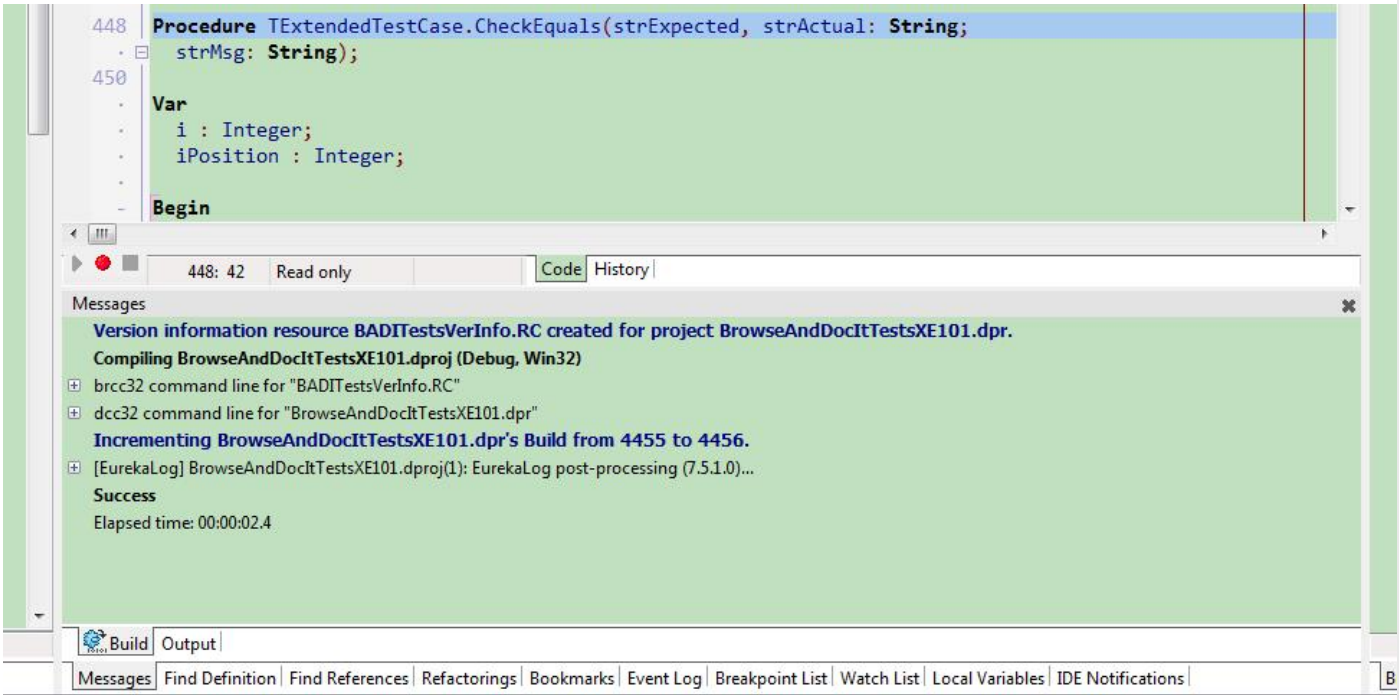
I had an itch the OTA couldn't scratch…

By David | March 9, 2017 0 Commen

Overview

So there I am, on the train and wondering why the dock areas at the side of RAD Studio can be hidden and auto pop out if you move the mouse over them or use a keyboard shortcut but the bottom dock area doesn't.

Why was I wondering this… well my day job consumes between 13 to 14 hours of my day and I've been getting too tired to do anything once I get back home so I thought I would install RAD Studio on the computer I use on the train which has a much smaller screen than the one at home and do some work while commuting. So I have the screen layed out with Browse and Doc It on the left, visible, things like the Project Manager, Object Inspector hidden on the right and the Message View docked with Find results, Refactoring results, etc at the bottom and I wanted the Message View to go away either by shortcut or automatically after a successful compile.



Can this all be done with the Open Tools API? Errr… not quite!

The Solution

The Open Tools API has the ability to display the Message View if you were to bind it to a keyboard shortcut but unfortunately there is no method of the `IOTAMessageServices` to do the converse and hide it. To make things harder, I wanted a particular behaviour in that if the Message View is docked in a tab set then I wanted the tab set to hide / show but if docked on its own I wanted just the Message View to hide / show.

You're probably thinking that this is game, set and match? Well no. Since an Open Tools API plug-in runs inside the IDE then you have access to the internals of the IDE. Now one thing to bare in mind is that the internals of the IDE (I'm limiting myself to the RAD Studio 2010+ Galileo IDEs) are not documented and can be changed between releases by Embarcadero without notice.

So how on earth do you go about fiddling with the internals of the IDE when you know nothing about it? You use something like my Delphi IDE Explorer. I wrote this a VERY long time ago for Delphi 3 and then forgot about it for a long time. Early last year when I came back to coding in general and the Open Tools API more specifically, I decide to update a few things. It was originally sparked by me doing

an internet search of my name and / or email address, which I do periodicaly, to see if here are any nasties on the internet like leaked passwords (which I've found before!). Anyway I found a link to Thomas Mueller's website where he had an updated version of my project which he was adding functionality to. Rudy Velthuis has something similar but not based on my code. What this made me do was do my own updates to get it working in Seattle but I also added the new RTTI functionality for RAD Studio 2010+ which gives you a load more information about the IDE than the previous versions. Anyway, it allows me to look around the IDE and see how things are constructed.

By trial and error I found out how the IDE behaviours when docking windows on their own or in tabs or floating, etc. What follows are the fruits of this exploration.

Adding an Action for the Keyboard Shortcut

I decided to add an Action to the IDE for the keyboard shortcut rather than an keyboard binding. A keyboard binding is only available if the editor has focus but I wanted to have the shortcut available whereever I was in the IDE. I decide on the shortcut SHIFT+ALT+M as this (according to GExperts) was not in use by the IDE. I didn't add a menu or an image to the action as it was not needed. There is a menu item for showing the Message View in the View menu and I initially thought I could find that action and add the shortcut but that wouldn't work as it only shows the Message View.

So in my main wizard's constructor I create an action as follows:

```
Constructor TMVHWizard.Create;

Var
  NService: INTAServices;

Begin
  ...
  FToggleMsgViewAction := Nil;
  If Supports(BorlandIDEServices, INTAServices, NService) Then
    Begin
      FToggleMsgViewAction := TAction.Create(NService.ActionList);
      FToggleMsgViewAction.ActionList := NService.ActionList;
      FToggleMsgViewAction.Name := 'DGHMsgViewHelperToggleMessageView';
      FToggleMsgViewAction.Caption := 'Toggle Message View';
      FToggleMsgViewAction.OnExecute := ToggleMessageViewAction;
      FToggleMsgViewAction.ShortCut := TMVHOptions.MVHOptions.MessageViewShortCut;
      FToggleMsgViewAction.Category := 'OTATemplateMenus';
    End;
  ...
End;
```

The above technique for adding actions to the IDE with the Open Tools API has been written about in Chapter 15: IDE Main Menus. One thing you MUST do is hold a reference so that your destructor can free the action and be a good citizen of the IDE else you'll leak memory in the first instance or get errors when loading / unloading the package repeatedly. The destructor is very simple:

```
Destructor TMVHWizard.Destroy;

Begin
  ...
  If FToggleMsgViewAction <> Nil Then
    FToggleMsgViewAction.Free;
  ...
  Inherited Destroy;
End;
```

Finally the notifier event that is called by the action is defined as follows:

```
Procedure TMVHWizard.ToggleMessageViewAction(Sender: TObject);

Begin
  ToggleMessageView;
End;
```

`ToggleMessageView` is a funtion that I've written and I'll explain it and it's associated functions in a while.

Hiding on Successful Compilations

So the next thing I wanted to do was hide the message view on successful compilation. I wanted to only close the view if the compilation was successful and the messages didn't contain errors or warnings but the Open Tools API doesn't expose the messages themself so this later criteria cannot be currently done. I do have an idea to resolve this as the Message View is a form of `TBaseVirtualTree`, one of my faviourite controls however I can't just get a reference and cast it as this is an IDE specific version. What I need to find is where the unit is defined in the IDE and see if I can add it to my project at design time to allow me access to the treeview.

Anyway, for the successful compilation of a project I decided to use the `IOTACompileNotifier` which I've written about before in Notify me of everything... – Part 1. I only need the `ProjectGroupCompileFinished` method to know if the project(s) have been compiled successfully. So I defined the following implementation:

```
Procedure TMVHCompileNotifier.ProjectGroupCompileFinished(Result: TOTACompileResult);

Begin
  If (Result = crOTASucceeded) And (mvhoEnabled In TMVHOptions.MVHOptions.Options) Then
    Begin
      FTimer.Interval := TMVHOptions.MVHOptions.HideMessageViewDelay;
      FTimer.Enabled := True;
    End;
End;
```

The method checks that the compilation(s) was successful and that the options to automatically close the Message View is enabled, then it sets the timer interval and starts the timer. The timer event handler is as follows:

```
Procedure TMVHCompileNotifier.CloseMessageView(Sender: TObject);

Begin
  FTimer.Enabled := False;
  HideMessageView;
End;
```

It disables the timer and calls a method to hide the Message View. Currently no checks are made for changing desktops between timer start and finish or if a second compilation is started before the timer expires but I'll do something about those at a later date.

"Open Tools API" Functions

I've placed all the "interesting" funtions is a unit called `MsgViewHelper.OTAFunctions.pas`. Although these are not all `IOTAXxxxx` interface related they to muck about with the IDE so I thought the name still applied.

The following functions all contribute to hiding and showing the Message View however not all are exposed by the unit for calling.

FindForm

This first function has the sole purpose of finding the Message View form in the IDE's list of forms. When you look into the IDE (and for that matter your own applications) you will find all the created forms in a list of forms in the `Screen` global variable. This method iterates through them looking for a form with the given name and return's its reference else if not found returns `nil`. Although this is a multiple purpose function, in this application is is used only to find the Message View form.

```
Function FindForm(Const strFormName : String) : TForm;

Var
  iForm: Integer;

Begin
  Result := Nil;
  For iForm := 0 To Screen.FormCount - 1 Do
    If CompareText(strFormName, Screen.Forms[iForm].Name) = 0 Then
      Begin
        Result := Screen.Forms[iForm];
        Break;
      End;
End;
```

IsDockableClassName

This method checks the given TWinControl's classname against an array of classnames which are the classname's of various docksites in the IDE. I check the classname rather than the control name as some of these are created dynamically. I identified the names of the dock site classes by inspecting the IDE structure with my Delphi IDE Explorer.

```
Function IsDockableClassName(Const P : TWinControl) : Boolean;

Var
  iDockClsName: Integer;

Begin
  Result := False;
  For iDockClsName := Low(strDockClsNames) To High(strDockClsNames) Do
    If CompareText(P.ClassName, strDockClsNames[iDockClsName]) = 0 Then
      Begin
        Result := True;
        Break;
      End;
End;
```

FindDockSite

This method tries to find the docksite in which the Message View form is docked. It walks backwards through the list of parent controls looking for a control with a classname that matches one of the known docksite classnames. If found the docksite reference is returned. A `TWinControl` is returned rather than a form because when a form is docked on its own it is placed in a panel not a descendant of a `TForm`.

```
Function FindDockSite(Const SourceControl : TWinControl) : TWinControl;

Var
  P : TWinControl;

Begin
  Result := Nil;
  P := SourceControl;
  While Assigned(P) Do
    Begin
      If IsDockableClassName(P) Then
        Begin
          Result := P;
          Break;
        End;
      P := P.Parent;
   End;
End;
```

IsMessageViewFocused

I tried a number of ways to determine whether the Message View was the focused control or not and ended up using the below. Why did I need this? Well the behaviour I wanted is that not only did I want the shortcut to hide / show the Message View but if the Message View was visible but not focused (in another tab or you are in the editor) then I wanted the focus to change to the Message View and only hide the Message View if it was active.

I settled on this as it was the only reliable way to determine this. It uses the IDE's main form's `ActiveControl` property and check whether its classname is equal to a specific treeview. I checked in previous IDE's back to 2010 to ensure that this had not changed. When you are doing things like this you cannot assume all IDEs are equal you must check that your assumptions hold true for ALL IDEs you are going to support and obviously it could be broken by a change in the future. If the `ActiveControl` has the classname expected then the Message View is focused.

```
Function IsMessageViewFocused : Boolean;

Var
  strActiveControl: String;
```

```
Begin
  Result := False;
  If Assigned(Application.MainForm.ActiveControl) Then
    Begin
      strActiveControl := Application.MainForm.ActiveControl.ClassName;
      Result := CompareText(strActiveControl, 'TBetterHintWindowVirtualDrawTree') = 0;
    End;
End;
```

IsMessageViewVisible

This method is the heart of this application and requires some explanation. Its a little long for my liking but any further refactoring in my mind would prevent someone from understnding it in full. The function returns a set of enumerates that define whether the Message View is visible and whether it has focus. The function needs to cater for floating windows as well has being docked to a tabbed docksite or a panel.

The first thing the function does it find the Message View form. If found we can process, if not we do nothing. Note: we should always find the Message View form.

Next, if the form is floating on its own we can determine the visibility and focusedness (is that a real word???) by calling a few functions of the form. If the form is docked (not floating) then we have a bit more work to do.

Using the form reference we need to find the docksite as what we do here depends upon whether the docksite is a tabbed form or a panel.

If its a panel (docked on its own adjacent to other windows) then we check whether the form is visible and determine its focusedness using the above function. Why did I check the visibility of the form NOT the panel? Well I found that when docked in a panel to hide and show the Message View I needed to hide and show the form rather than the panel as hiding the panel hid the whole docksite rather that the panel and that was not the behaviour I wanted.

If we are docked in a tabbed docksite (which is a form) I determine the visbility of the docksite as a form and the focusedness as before. Hiding and showing the tabbed docksite provide the behaviour I wanted else you would just hide and show the Message View from the tab set.

```
Function IsMessageViewVisible(Var Form : TForm; Var DockSite : TWinControl) : TMsgViewStates;

Begin
  Result := [];
  Form := FindForm(strMessageViewForm);
  DockSite := Nil;
  If Assigned(Form) Then
    Begin
      If Form.Floating Then
        // If floating
        Begin
          If Form.Visible Then
            Include(Result, mvsVisible);
          If Form.Active Then
            Include(Result, mvsFocused);
        End Else
        // If Docked
        Begin
          DockSite := FindDockSite(Form);
          If DockSite Is TWinControl Then
            Begin
              // If docked to a panel we don't want to hide the panel but the message window.
              If DockSite Is TPanel Then
                Begin
                  If Form.Visible Then
                    Begin
                      Include(Result, mvsVisible);
                      If IsMessageViewFocused Then
                        Include(Result, mvsFocused);
                    End;
                  DockSite := Nil;
```

```
                End Else
                // If docked to a tabset we do want to hide the dock tabset
                Begin
                  If DockSite.Visible Then
                    Begin
                      Include(Result, mvsVisible);
                      If IsMessageViewFocused Then
                        Include(Result, mvsFocused);
                    End;
                End;
            End;
        End;
    End;
End;
```

ShowMessageView

This method shows the Message View. You will note that it uses the Open Tools API to do this but this alone would not focus the Message View if it was already visible in the IDE, so I needed to add focusing the form. However when writing the above function I found that I was calling the same code for showing the form / docksite before each call to this method so I moved those calls into this method.

```
Procedure ShowMessageView(Form : TForm; DockSite : TWinControl);

Var
  MsgServices: IOTAMessageServices;

Begin
  If Assigned(DockSite) Then
    DockSite.Show
  Else
    Form.Show;
  Form.SetFocus;
  If Supports(BorlandIDEServices, IOTAMessageServices, MsgServices) Then
    MsgServices.ShowMessageView(MsgServices.GetGroup('Build'));
End;
```

ToggleMessageView

This method, using the information from above regarding form, docksite, visibility and focusedness either hides, shows or focuses the Message View. This method is the one called by the Action installed into the IDE.

```
Procedure ToggleMessageView;

Var
  Form: TForm;
  DockSite: TWinControl;
  MsgViewStates : TMsgViewStates;

Begin
  MsgViewStates := IsMessageViewVisible(Form, DockSite);
  If Assigned(Form) Then
    If mvsVisible In MsgViewStates Then
      Begin
        If mvsFocused In MsgViewStates Then
          Begin
            If Assigned(DockSite) Then
              DockSite.Hide
            Else
              Form.Hide;
          End Else
            ShowMessageView(Form, DockSite);
```

```
      End Else
        ShowMessageView(Form, DockSite);
End;
```

HideMessageView

Finally this is the method called by the compile notifier timer and it simply hides the Message View depending upon whether its docked or floating.

```
Procedure HideMessageView;

Var
  MsgViewState: TMsgViewStates;
  Form : TForm;
  DockSite: TWinControl;

Begin
  MsgViewState := IsMessageViewVisible(Form, DockSite);
  If Assigned(Form) Then
    If Assigned(DockSite) Then
      DockSite.Hide
    Else
      Form.Hide;
End;
```

I don't know whether this plug-in will be useful to others as it might just be a pecularity of the way I work but I hope that it provides people who are looking into the Open Tools API with ideas when they find that the API does not do all they want.

Downloads

The compiled BPLs and source code for this plug-in can be found on the web page Message View Helper which contains links to a downloadable ZIP file with the BPLs and source code or a GitHub link to the source code.

Related posts:

1. Another itch the OTA couldn't scratch… (20)
2. Notify me of everything… – Part 1 (9.1)
3. Chapter 5: Useful Open Tools Utility Functions (8)
4. Chapter 8: Editor Notifiers (7.6)
5. The Open Tools API using C++ Builder (7.1)

Category: IDE Explorer Open Tools API RAD Studio Tags: INTAServices, IOTACompileNotifier, IOTAMessageServices, IOTAWizard, OTA, RAD Studio