Dave's Development Blog

Software Development using Borland / Codegear / Embarcadero RAD Studio

Chapter 6: Open Tools API Custom messages

By David | May 1, 2011 0 Comment

In my last post on OTA I showed some methods for handling messages in the Open Tools API. What I didn't cover in the post was custom messages.

I use custom message in my Integrated Testing Helper IDE plug-in to colour message depending upon the success or failure of the command-line tools being run. This way the user gets immediate feedback when looking at the messages as to what has failed.

There are four interface for custom messages. These are **IOTACustomMessage**, **IOTACustomMessage50**, **IOTACustomMessage100** and **IOTACustomDrawMessage**. The first three are to do with the management of message information and the fourth (and the more interesting one for me) deals with custom drawing the message.

The first interface **IOTACustomMessage** defines properties and method for handling line numbers, column numbers, file name and source text for message that require associations with code in the IDE. The second interface **IOTACustomMessage50** defines properties and methods for dealing with nested messages (will return to the before the end of this chapter). The third interface **IOTACustomMessage100** provides methods to allow a message to override the default IDE mechanism for navigating to a file and allowing the message to do this instead. The fourth interface **IOTACustomDrawMessage** allow the message to draw itself allowing for a change of font name, style, size and colour (you can also change the background).

While its in my mind, there is one thing to ensure your expert does when using custom message and that is clear those custom messages from the message windows and tabs before being unloaded. If you do not do this and say unload a BPL, your message view will cause access violation because it can't find the drawing code for the messages.

First we'll look at the definition of a custom message that's used in my Integrated Testing Helper IDE plug-in as below:

```
TDGHCustomMessage = Class(TInterfacedObject, IOTACustomMessage, INTACustomDrawMessage)
Private
  FMsg : String;
```

```
      FFontName   : String;
      FForeColour : TColor;
      FStyle : TFontStyles;
      FBackColour : TColor;
      FMessagePntr : Pointer;
    Protected
      function CalcRect(Canvas: TCanvas; MaxWidth: Integer; Wrap: Boolean): TRect;
      procedure Draw(Canvas: TCanvas; const Rect: TRect; Wrap: Boolean);
      function GetColumnNumber: Integer;
      function GetFileName: string;
      function GetLineNumber: Integer;
      function GetLineText: string;
      procedure ShowHelp;
      Procedure SetForeColour(iColour : TColor);
    Public
      Constructor Create(strMsg : String; FontName : String;
        ForeColour : TColor = clBlack; Style : TFontStyles = [];
        BackColour : TColor = clWindow);
      Property ForeColour : TColor Write SetForeColour;
      Property MessagePntr : Pointer Read FMessagePntr Write FMessagePntr;
    End;
```

Here we define a class that implements 2 of the 4 interfaces. Since I’m not interested in child message handling in this way (because it doesn't help me with earlier version of Delphi that don't have this functionality) or changing the default message handling then I'll concentrate on the first and the last interfaces.

The class we defined must implement all the methods of the interfaces referenced in the inheritance list, thus we have to implement methods for returning line, column, filename and source text for the first interface and methods to handling drawing for the second interface (fourth above). Additionally I've defined 2 more properties to allow me to change the colour of the message and manage the messages pointer reference (this reference is used for nesting the messages which we will tackle later). The reason for not using the **TCustomMessage50** interface for handling child messages is that this tool needs to work with earlier version of Delphi so I actual manage this information myself.

Below are the implemented interface method for the first interface:

```
  function TDGHCustomMessage.GetColumnNumber: Integer;
  begin
    Result := 0;
  end;

  function TDGHCustomMessage.GetFileName: string;
  begin
    Result := '';
  end;

  function TDGHCustomMessage.GetLineNumber: Integer;
```

```
begin
  Result := 0;
end;

function TDGHCustomMessage.GetLineText: string;
begin
  Result := FMsg;
end;

procedure TDGHCustomMessage.ShowHelp;
begin
  //
end;
```

Since I don't want to browse source code with these messages they simply return default information so that double clicking them has no effect.

Next, lets look at the constructor:

```
constructor TDGHCustomMessage.Create(strMsg: String; FontName : String;
  ForeColour : TColor = clBlack; Style : TFontStyles = [];
  BackColour : TColor = clWindow);

Const
  {$IFNDEF D2009}
  strValidChars : Set Of Char = [#10, #13, #32..#128];
  {$ELSE}
  strValidChars : Set Of AnsiChar = [#10, #13, #32..#128];
  {$ENDIF}

Var
  i : Integer;
  iLength : Integer;

begin
  SetLength(FMsg, Length(strMsg));
  iLength := 0;
  For i := 1 To Length(strMsg) Do
    {$IFNDEF D2009}
    If strMsg[i] In strValidChars Then
    {$ELSE}
    If CharInSet(strMsg[i], strValidChars) Then
    {$ENDIF}
      Begin
        FMsg[iLength + 1] := strMsg[i];
        Inc(iLength);
      End;
  SetLength(FMsg, iLength);
  FFontName := FontName;
  FForeColour := ForeColour;
```

```
    FStyle := Style;
    FBackColour := BackColour;
    FMessagePntr := Nil;
  end;
```

The constructor has a conditional define in it. This is simply to make the code work with different versions of Delphi as Delphi 2009's string were changed to UniCode. The constructor simply assigns the passed message to an internal string first removing any carriage returns or line feeds and extended characters (as the display as boxes) and initialises the custom message.

Now we can move onto the drawing code. The drawing code is in 2 parts, first the calculation of the size of the message to be drawn and then the drawing itself. We'll first look at the **CalcRect()** method as below:

```
  function TDGHCustomMessage.CalcRect(Canvas: TCanvas; MaxWidth: Integer;
    Wrap: Boolean): TRect;

  begin
    Canvas.Font.Name := FFontName;
    Canvas.Font.Style := FStyle;
    Result:= Canvas.ClipRect;
    Result.Bottom := Result.Top + Canvas.TextHeight('Wp');
    Result.Right := Result.Left + Canvas.TextWidth(FMsg);
  end;
```

First the font name and font style are assigned to the messages canvas reference, then canvas rectangle is obtained and adjusted to suit the height and width of the message with the given font and style.

Next the actual drawing code as follows:

```
  procedure TDGHCustomMessage.Draw(Canvas: TCanvas; const Rect: TRect;
    Wrap: Boolean);

  begin
    If Canvas.Brush.Color = clWindow Then
      Begin
        Canvas.Font.Color := FForeColour;
        Canvas.Brush.Color := FBackColour;
        Canvas.FillRect(Rect);
      End;
    Canvas.Font.Name := FFontName;
    Canvas.Font.Style := FStyle;
    Canvas.TextOut(Rect.Left, Rect.Top, FMsg);
  end;
```

First the colours are assigned and the background rectangle rendered then the canvas is set with the message's font name and style and finally the messages is drawn on the canvas. Note that I do not use the **wrap** parameter. You could wrap you messages to fit in the window width but to do this you would then have to change the **CalcRect()** method and the **Draw()** method to first calculate the height of the wrapped message with the Win32 API **DrawText()** method and then draw the message with the same API call.

Now we have a custom message we need a way to add these messages to the message window. I do this with the following procedure:

```
Function AddMsg(strText : String; boolGroup, boolAutoScroll : Boolean;
  FontName : String; ForeColour : TColor; Style : TFontStyles;
  BackColour : TColor = clWindow; Parent : Pointer = Nil) : Pointer;

Var
  M : TDGHCustomMessage;
  G : IOTAMessageGroup;

begin
  With (BorlandIDEServices As IOTAMessageServices) Do
    Begin
      M := TDGHCustomMessage.Create(strText, FontName, ForeColour, Style, BackColour);
      Result := M;
      If Parent = Nil Then
        Begin
          G := Nil;
          If boolGroup Then
            G := AddMessageGroup(strITHelperGroup)
          Else
            G := GetMessageGroup(0);
          {$IFDEF D2005}
          If boolAutoScroll <> G.AutoScroll Then
            G.AutoScroll := boolAutoScroll;
          M.MessagePntr := AddCustomMessagePtr(M As IOTACustomMessage, G);
          {$ELSE}
          AddCustomMessage(M As IOTACustomMessage, G);
          {$ENDIF}
        End Else
          {$IFDEF D2005}
          AddCustomMessage(M As IOTACustomMessage, Parent);
          {$ELSE}
          AddCustomMessage(M As IOTACustomMessage);
          {$ENDIF}
    End;
end;
```

Firstly, this method is design to work in multiple version of Delphi, hence the conditional compilation. Delphi 2005 and above support nested messages but earlier version do not. Additional, the IDE handles nested messages in a way that's not what you would expect. The methods to add a message return a pointer to the message NOT a reference to the custom message class and its this pointer you need to nest messages.

The method creates the custom message. It will also assigns the message to the ITHelper's message group (remember this is code from ITHelper). Finally it adds the message to the IDE using the appropriate message method. Some of the message adding methods take the parent point and some do not. I assign the messages pointer value to the message's MessagePntr property so that it can be referenced later IF I want to nest more messages.

So there it is. If you want to see more on this subject then please download and browse the source code for the ITHelper IDE plugin.

regards
Dave.

Category: Integrated Testing helper Open Tools API Tags: Borland, BorlandIDEServices, CodeGear, Delphi, Embarcadero, Experts, IOTACustomDrawMessage, IOTACustomMessage, IOTACustomMessage100, IOTACustomMessage50, IOTAMessageServices, ITHelper, OTA, RAD Studio