April, 2006

Enhancing the Effects Listener

An attempt to extend a sample leads to a better understanding of report listeners.

By Tamar E. Granor, technical editor

Recently, I wrote a piece in the Advisor Answers column showing how to use the EffectsListener class to specify dynamic formatting of some elements in a report (http://My.Advisor.com/doc/17550). I mentioned that the class could be subclassed to handle additional effects. When I actually tried doing so, I ended up learning a lot more than I expected to about the architecture of reports and report listeners. This article shares that experience with you.

EffectsListener is a report listener that comes with VFP 9. You'll find it in Samples\Solutions\Europa\DynamicFormatting.PRG. As I explained originally, you can put directives into the User data for a report field in this form:

```
*:EFFECTS <effect> = <expression>
```

When you run the report with the EffectsListener hooked in, those directives are read and handled, letting you use data to decide the color of an item or its formatting.

EffectsListener supports two effects: ForeColor and Style. I decided to subclass it to provide a dynamic background color for report fields, as well.

The effects classes

Three classes collaborate to produce the desired effects: EffectsListener, EffectObject, and EffectHandler. Figure 1 shows the classes in DynamicFormatting.PRG in the Class Browser.

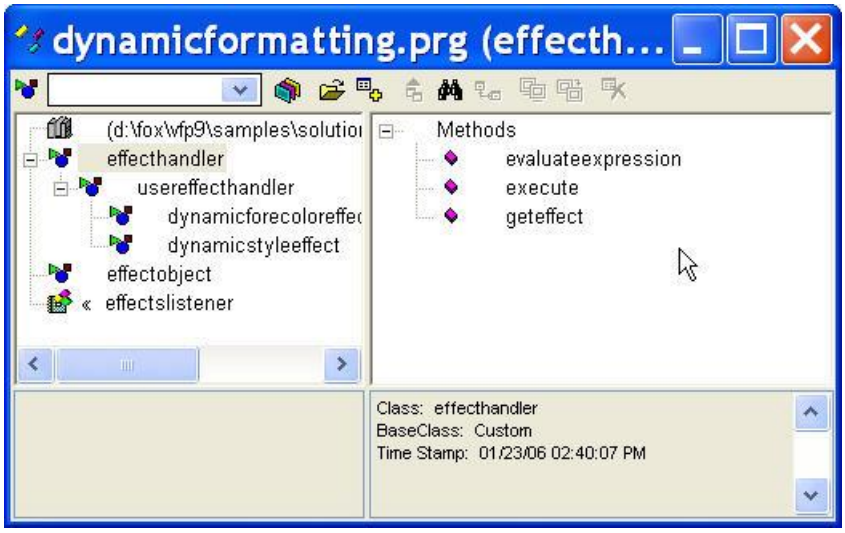

Figure 1: Creating dynamic effects—To create dynamic effects in reports, the report listener uses three class hierarchies.

I'll start with EffectObject because it's the simplest. Here's the entire definition of the class in DynamicFormatting.PRG:

```
define class EffectObject as Custom
  oEffectHandler = .NULL.
  cExpression    = ''
enddefine
```

This class serves as a placeholder for the name of an effect and the expression that evaluates it. Subclasses of EffectHandler instantiate EffectObject and store the specified information; when processing the report, EffectsListener and its subclasses build a collection of EffectObject objects for each item in the report. Most importantly, you don't have to subclass EffectObject to add effects; just use it.

The class that does most of the work of effects is EffectHandler. It has three custom methods:

GetEffect—This class creates an EffectObject object and puts the appropriate data into it. It's specified in the UserEffectHandler subclass of EffectHandler and you shouldn't need to write any code for it.

ExecuteExpression—This class is defined in EffectHandler itself; it simply evaluates the expression it's passed and returns the result. Again, you shouldn't need to do anything with this method.

Execute—This is the method you need to override for your special effect. It receives two parameters: an object containing formatting

characteristics of the current report item and the expression for that item.

One of the classes provided is DynamicForeColorEffect; I copied that class as a starting point for my DynamicBackColorEffect class. Here's the result, after I made the necessary changes:

```
DEFINE CLASS DynamicBackColorEffect AS ;
  UserEffectHandler OF ;
  HOME(2)+"Solution\Europa\DynamicFormatting.PRG"
cEffectName = "BACKCOLOR"
lAdjustSize = .T.
nTextWidth = 0
FUNCTION Execute(toObjProperties, tcExpression)
local lnColor, lnFillRed, lnFillGreen, lnFillBlue

lnColor = This.EvaluateExpression(tcExpression)
if vartype(lnColor) = 'N'
  lnFillRed   = bitand(lnColor, 0x0000FF)
  lnFillGreen = bitrshift(bitand(lnColor, 0x00FF00),  8)
  lnFillBlue  = bitrshift(bitand(lnColor, 0xFF0000), 16)
  with toObjProperties
    if .FillRed <> lnFillRed or ;
      .FillGreen <> lnFillGreen or ;
      .FillBlue <> lnFillBlue
      .FillRed   = lnFillRed
      .FillGreen = lnFillGreen
      .FillBlue  = lnFillBlue
      .FillAlpha = 255
      .Reload   = .T.
    endif .FillRed <> lnFillRed ...
  endwith
endif vartype(lnColor) = 'N'
endfunc
ENDDEFINE
```

I didn't have to make many changes. I changed cEffectName to "BACKCOLOR", so the directive for using this effect is:

```
*:EFFECT BACKCOLOR = <expression>
```

DynamicForeColorEffect operates on the PenRed, PenGreen, and PenBlue properties. I changed the code to refer to the FillRed, FillGreen, and FillBlue properties respectively, so that I'm working on the object's BackColor rather than ForeColor. Although the original code used a different technique than I would have used to parse the RGB value into its components, I saw no reason to change code that worked. Setting the

Reload property of the toObjProperties object to .T. tells the report listener that data has been changed, so it can take appropriate action.

The third class hierarchy involved has EffectsListener as its root. You need to subclass that class to add effects. However, in most cases, the only method you need to extend is the Init method. The Init method of the original EffectsListener class contains this code:

```
function Init
  dodefault()
  with This
    .oEffectHandlers = createobject('Collection')
    .oEffectHandlers.Add( ;
       createobject('DynamicForeColorEffect'))
    .oEffectHandlers.Add( ;
       createobject('DynamicStyleEffect'))
  endwith
endfunc
```

To add effects, you need to add the appropriate subclass of EffectHandler to the oEffectHandlers collection. To add a new handler, just instantiate it in the Init. Here's the code from the Init of my MoreEffects subclass:

```
PROCEDURE Init
DODEFAULT()
This.oEffectHandlers.Add( ;
  CREATEOBJECT("DynamicBackColorEffect"))
RETURN
```

This is all it takes to change the back color of an item based on an expression. In the report shown in figure 2, the User Data of the Country item contains this (without the extra returns needed here for formatting):

```
*:EFFECTS BackColor =
  IIF(Country <> "USA", rgb(255,0,0), rgb(255,255,255))
*:EFFECTS ForeColor =
  IIF(Country<>"USA", rgb(255,255,255), 0)
```

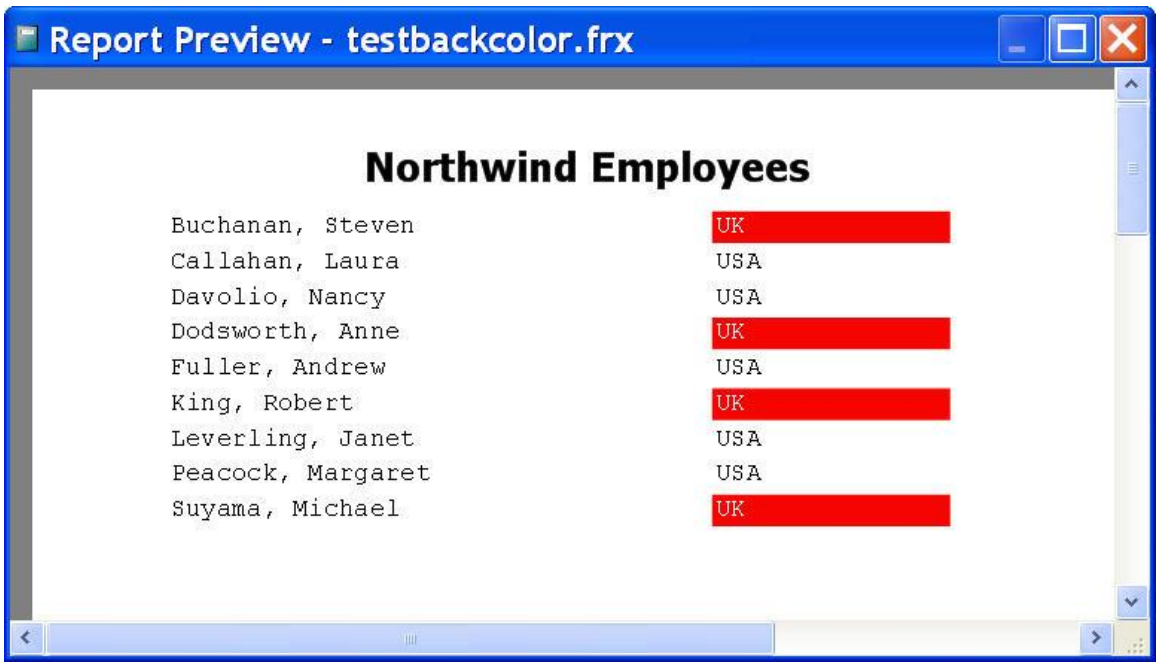

Figure 2: Dynamic background color—This report shows the DynamicBackColorEffect handler in action.

At this point, I could have stopped, but I didn't like the way the background color filled the entire space allocated for the field rather than only the area used. So I started my odyssey into understanding much more about how report listeners interact with reports.

Computing the right size

The first problem was figuring out how wide the field should be, based on the data to be printed. All the information I needed to compute the size—the text itself, plus the font face, size, and style—was available in the Execute method of DynamicBackColorEffect, but there's no way to directly affect the size of the field from that method.

A little exploration convinced me, though, that I wouldn't have another opportunity to compute the field width. So I added two properties to the DynamicBackColorEffect class:

lAdjustSize—This property is a flag that indicates whether the size of a given field should be changed. The code does so only if the background color has been changed.

nTextWidth—This property holds the new width once it's computed.

Reports work with units of 1/960 of an inch, so conversion is needed. I added a method called ComputeWidth to the class to handle the job. Here's the code in ComputeWidth:

```
PROCEDURE ComputeWidth(cString, cFontName, ;
                       nFontSize, nFontStyle)
* Compute the width of the specified string rendered
* in the specified font. Return the value in the
* units used by reports (1/960 of an inch).
LOCAL cFontStyle, nWidth
cFontStyle = IIF(BITTEST(nFontStyle, 0), "B", "") + ;
             IIF(BITTEST(nFontStyle, 1), "I", "")
* First, get width in chars
nWidth = TxtWidth(ALLTRIM(cString), cFontName, ;
                  nFontSize, cFontStyle)
* Now, multiply by average char width
nWidth = nWidth * ;
         FONTMETRIC(6, cFontName, nFontSize, cFontStyle)
* Pixel = 1/72 of an inch. Convert to report units
nWidth = nWidth / 72 * 960
RETURN nWidth
```

I modified DynamicBackColor's Execute method to call ComputeWidth and store the result by adding these lines inside the IF statement:

```
This.lAdjustSize = .T.
This.nTextWidth = This.ComputeWidth(.Text, .FontName, ;
   .FontSize, .FontStyle)
```

Changing the field size

The real challenge was figuring out how to change the actual printed size of the field. Doing so forced me to explore the _ReportListener class in HOME()+"FFC_ReportListener.VCX," from which EffectsListener is subclassed, as well as to dig into the available documentation for report listeners.

The report listener method that displays a field is called Render; it's the last opportunity to make changes before the object is drawn and, in fact, you can take over rendering a field yourself in this method. I didn't want to do that, though; I just wanted to change the size of the rendered object.

Render receives a number of parameters, including the left, top, width, and height of the item to be displayed. So the key to changing the displayed width is to change Render's width parameter then let the built-in behavior take over.

Reports are stored in FRX files, which are just tables with a special extension. EffectsListener has an array property, aRecords, with one row for each record in the report's FRX. Early on, EffectsListener populates the second column of aRecords with a collection of the effects handlers that apply to that record. The Render method of MoreEffects uses that array to determine if the size of the current item can be adjusted. If so, it calls a custom method of DynamicBackColorEffect, AdjustObjectSize, to make the change. Finally, it uses DoDefault to call the default behavior of Render, passing the changed value, and NODEFAULT to prevent the object from being rendered without the change.

```
PROCEDURE Render
LPARAMETERS m.nFRXRecno, m.nLeft, m.nTop, ;
            m.nWidth, m.nHeight, ;
            m.nObjectContinuationType, ;
            m.cContentsToBeRendered, m.GDIPlusImage
LOCAL loEffect, loHandler
IF VARTYPE(This.aRecords[m.nFRXRecno, 2])= "O"
   for each loEffect in This.aRecords[m.nFRXRecno, 2]
      loHandler = loEffect.oEffectHandler
      IF PEMSTATUS(loHandler, "AdjustObjectSize", 5)
         loHandler.AdjustObjectSize( ;
           @m.nLeft, @m.nTop, @m.nWidth, @m.nHeight)
       ENDIF
   next loEffect
ENDIF
DODEFAULT(m.nFRXRecno, m.nLeft, m.nTop, ;
          m.nWidth, m.nHeight, ;
          m.nObjectContinuationType, ;
          m.cContentsToBeRendered, m.GDIPlusImage)
NODEFAULT
RETURN
```

DynamicBackColorEffect's AdjustObjectSize method is simple. It checks the lAdjustSize flag and, if it's true, changes the Width parameter. The method accepts all four position- and size-related parameters, so you can use it for broader changes, if appropriate.

```
PROCEDURE AdjustObjectSize(m.nLeft, m.nTop, ;
                           m.nWidth, m.nHeight )
* If the backcolor has been changed, the size
* of the object should be adjusted so the backcolor
* block isn't bigger than the content.
IF This.lAdjustSize
   m.nWidth = This.nTextWidth
   This.lAdjustSize = .F.
ENDIF
RETURN
```

Figure 3 shows the report with these changes to the report listener. This issue's Professional Resource CD includes MoreEffects.PRG, which contains the MoreEffects subclass of EffectsListener, and the DynamicBackColorEffects class; the PRD also includes TestBackColor.FRX, the report shown in figures 2 and 3, and TestBackColor.PRG, a simple program to instantiate the report listener and run the report.

Figure 3: Resized field—Getting the dynamic backcolor right required changing the size of the item to be rendered.

What next?

There isn't too much more you can do in the way of dynamic effects using the architecture EffectsListener provides. The toObjProperties parameter passed to the Execute method has only a few properties that aren't covered by one of the existing effects handlers. You could provide dynamic choice of FontName or FontSize; in both cases, you would need to use the resizing capabilities described in this article, since changing font face or font size is likely to change the width of a field. Be careful in introducing this sort of change because reports containing too many font faces or sizes tend to look more like ransom notes than business documents.

Sidebar: Why UserEffectHandler?

In looking at the class hierarchy in figure 1 of the main article, you might have wondered why it includes the UserEffectHandler subclass

of EffectHandler. Why not subclass the specific effects directly from EffectHandler?

I didn't design this hierarchy, but I think the idea is that there might be more than one way to specify dynamic effects. UserEffectHandler includes the code that reads the User memo field from the report and parses what it finds. A different subclass of EffectHandler might look elsewhere for that information.