A New Beginning

Doug Hennig

The development world is moving to reusable components. As goes the world, so goes Doug's column. We'll start off the new "Best Tools" column with SFThermometer, a handy progress meter class.

For several years now (coinciding with my first look at Visual FoxPro 3), I've felt that the role of software developers was going to evolve. Instead of writing reams of hand-crafted code for each new application, I believe we're going to be (and more and more, already are) combining reusable components together and mostly focusing on providing the links and communication between those components. While I've always tried to create library routines that could be reused whenever possible, the object-oriented nature of VFP makes it so much easier to create discrete, independent components that can be pieced together to build larger modules. Add in the thousands of ActiveX controls available and OLE Automation with applications like Word and Excel, and we can now create modules that have almost no old style code for making pieces work together. It's like the software version of the toolbox in my basement: each tool does just one job, but does it very well. I can use various combinations of these tools together to get any job done (well, that exaggerates my handy-man skills, but that's not the fault of the tools).

The articles I've written in the past year and a half for the "Best Practices" column were somewhat of a mixed bag. Some were ideas or tidbits columns, while others presented a solution to a particular problem. Some articles presented code that you could just drop into an application while others were more conceptual than "plug 'n' play". After sharing our views on reusable components, FoxTalk editor Whil Hentzen and I agreed that this column should change its focus. Starting with this month's issue, I'm going to concentrate on providing reusable components that you can include in your applications with as little effort as possible. In other words, it's now a "Best Tools" column.

Here's a map of where we're going:

- All objects will be based on the subclassed VFP base classes described in the "PEMs for Your Base Classes" article in the February 1997 issue of FoxTalk. I've renamed the VCX file these subclasses exist in from CONTROLS.VCX to SFCTRLS.VCX because we use this VCX at Stonefield for production applications and start all our VCX names with "SF" to avoid conflicts with other third party tools. Because we tweak SFCTRLS.VCX from time to time to improve the classes or add new behavior, this VCX will be provided along with each month's component. As a result, I don't recommend making any changes in any of the SFCTRLS classes or your changes will be overwritten when the next month's version comes out. If you have any ideas you'd like to see incorporated in these parent classes, please let me know.
- Other VCXs will contain specialized components. For example, SFMOVER.VCX will contain classes used in a generic two-column mover control, SFMGRS.VCX will contain "manager" classes, and SFCCTRLS.VCX will contain miscellaneous

custom controls. Currently, SFCCTRLS.VCX contains the SFMemoDictEditBox, SFAutoFillComboBox, and SFVisibleEditTextBox classes described in the "Custom Classes" article in the March 1997 issue of FoxTalk. Like SFCTRLS.VCX, SFCCTRLS.VCX will change with time (in fact, it'll change a lot more frequently); we'll add new classes to it and possibly improve existing classes.

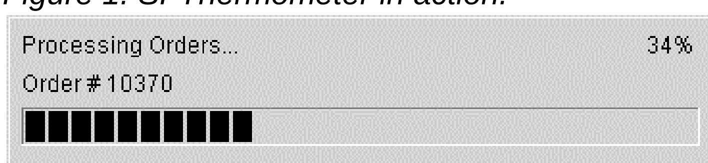
- My goal is to create tools you can just drop on a form or instantiate using CREATEOBJECT(). The ultimate goal is to make them all standalone and independent so they'll work with any other objects in any framework (commercial or one you build yourself). I may not always achieve this goal (we may need some interaction between classes), but I'll try to minimize this as much as possible.
- In addition to objects I've created, I'll also discuss great tools by other folks. If you have a component you'd like to share, let me know about it.

SFThermometer

We'll start the new direction off this month by creating a thermometer bar object. Thermometers (also known as progress bars) are used to display the progress of a lengthy process so the user can judge how much more time is required and to show that the application isn't hung up. I use a class called SFThermometer in lots of places, including to display the progress of the creation of Word documents and to show how a time-consuming import job is proceeding.

SFThermometer, shown in Figure 1, uses the ProgressBar ActiveX control (one of the ActiveX controls that comes with VFP 5) to do the majority of the work. However, we'll add some additional functionality as you'll see.

Figure 1. SFThermometer in action.



In addition to the ProgressBar control, SFThermometer has three labels: one for the "main" label (the overall job being done, such as posting transactions), one for the "current task" (a subset of the job, such as calculating totals), and one to show the current percentage complete. These labels allow you to show what stage the process is at. It also has a Cancel button that allows the user to cancel the process. We'll see how that's accomplished later.

I originally planned to base SFThermometer on a form (SFForm) but later changed my mind and based it on SFContainer (a subclass of Container) instead. While this design has the drawback that SFThermometer has to be dropped on a form before it can be used, the advantage is that the progress bar doesn't have to appear in a standalone form. You can include it in any form that might benefit from it. For example, you might have a dialog that displays processing options (such as transaction date range) and the progress meter. When the user clicks on the OK button, the progress meter is enabled and shows the progress of the processing.

SFThermometer has custom methods (SetTitle() and SetMaximum()) that set the caption for the main label and the maximum value for the ProgressBar control. To update the meter, use the Update() method. It accepts two parameters: the current value of the meter and the caption for the current task label in the form. The value passed to Update() is interpreted differently depending on the setting of the lPassPercent property of the class. If lPassPercent is .T. (the default is .F.), Update() expects that the value passed is a percentage; if not, it expects the value is an absolute value and calculates the percentage by dividing by the maximum value.

Here's how you use SFThermometer:

- Drop it on a form. You might also drop it on a form class and then instantiate that class whenever you need a progress bar.
- Somewhere in the form (or something that instantiates the form), call the SetTitle() method of SFThermometer to set the caption for the main label.
- If the value of the progress bar isn't a percentage but an absolute value (for example, if you have 1,573 records in a table to process and you increment a counter for each record processed), call the SetMaximum() method to set the maximum value for the control (1,573 in this example). If the value is a percentage, set the lPassPercent property to .T.
- Set the cCancelProperty property to the name of a variable (it could also be the name of a property of some object) that is normally .F. but will become .T. if the user clicks on the Cancel button. We'll see how this is used in a moment.
- Do your processing loop, calling the Update() method of SFThermometer whenever you want the progress bar updated (for example, just before the ENDSCAN in a SCAN loop). Pass Update() the value (absolute or percentage) for the meter and optionally the name of the current task being done.
- The processing loop should be defined to terminate when a certain variable or property of an object becomes .T. This provides a clean exit from the loop: if the user presses the Cancel button in the progress meter, the variable is set to .T., which causes the loop to be terminated naturally (that is, without an EXIT command). For example, I often use a variable called llStopped and use code such as the following for my loop:

```
llStopped = .F.
oProgress.cCancelProperty = 'llStopped'
scan while not llStopped
  * do some processing
  oProgress.Update(<value>)
endscan while not llStopped
```

PROGRESS.PRG is a sample program that shows how SFThermometer can be used. The PROGRESS form simply has an SFThermometer container dropped on it and PROGRESS.PRG runs this form to display the meter. It goes through the ORDERS table in the sample database that comes with VFP, adding up the price of each product ordered for each order. Since there are several thousand records, it takes a little while to process, so you can see the progress bar in action and see how it cleanly terminates when you press Cancel.

Class Anatomy

SFThermometer has the following custom properties:

- cCancelProperty (public): the name of a variable or property (including the complete reference to the object if necessary) to set to .T. when the Cancel button is clicked. If this property isn't filled in, the Cancel button is hidden.
- lPassPercent (public): .T. if the value passed to the Update() method is in percent or .F. if it's an absolute value.
- nPercent (protected): the last percentage displayed in the progress meter. This is used so the progress meter isn't updated if the percentage hasn't changed since the last time Update() was called.

This class has several custom methods (the code isn't shown for all the methods here, only those of special interest; download the files from the Subscriber Downloads to examine the code). As I mentioned earlier, SetTitle() sets the caption for title label and SetMaximum() sets the maximum value for the ProgressBar controls. Show() hides the Cancel button if the cCancelProperty isn't filled in and adjusts the width of the ProgressBar control to fit the size of the container. This allows you to resize an instance of SFThermometer for a certain form without having to resize all the controls inside the container.

Update() updates the progress meter with a new value and changes the caption of the current task label if a new caption is specified. It also calls the custom CheckCancel() method to determine if the user clicked on the Cancel button.

```
lparameters tnCurrent, ;
  tcTask
local lnPercent

* Don't let the percentage go over the maximum or
* under the minimum.

with This
  if .lPassPercent
    lnPercent = max(min(tnCurrent, .oProgress.Max), ;
      .oProgress.Min)
  else
    lnPercent = max(min(int(tnCurrent/.oProgress.Max * 100), 100), 0)
  endif .lPassPercent

* If the task to display was passed, display it.

  if not empty(tcTask)
    .lblTask.Caption = tcTask
  endif not empty(tcTask)

* If the percentage has changed, store the calculated
* percentage.

  if lnPercent <> .nPercent
    .nPercent           = lnPercent
    .lblPercent.Caption = ltrim(str(lnPercent)) + '%'
    .oProgress.Value    = tnCurrent
  endif lnPercent = .nPercent

* Let's ensure we're visible.
```

```
  if not .Visible
    .Show()
  endif not .Visible

* See if the Cancel button has been pressed.

  .CheckCancel()
endwith
```

SFThermometer uses a very interesting technique to permit a “hard” loop (such as your processing loop) to be interrupted. As you may have discovered if you’ve already tried to do this, VFP only sees the user trying to click on a Cancel button if it’s in a “wait” state; that is, when it’s awaiting action from the user and not executing any code. So, if you put a Cancel button on a form, any presses on the button are ignored while any code (such as a processing loop) is executing.

VFP 5 introduced a new command called DOEVENTS. Basically, this command adds a tiny wait state into executing code. It checks the Windows event queue to see if anything needs to be handled, handles it, then carries on. This sounded like a perfect way to detect if the user clicked on the Cancel button, so I originally used DOEVENTS near the end of the Update method. However, to my horror, I discovered this took an *order of magnitude* longer to process my loop!

Fortunately, Alan Schwartz and Mac Rubel suggested a better alternative: check to see if the mouse is over the Cancel button and if it’s been pressed (MDOWN() returns .T.). If so, only then use DOEVENTS to allow the click event to be processed (that is, the Cancel button visually gets pressed down). This is handled in the CheckCancel() method, which is called by Update().

CheckCancel() does the following: if the Cancel button is visible and the mouse is pressed, it checks to see if the user is clicking on the Cancel button using the IsMouseOverCancel() method. If so, it uses DOEVENTS to allow the Cancel button to appear to depress, then checks to see if the user let up the mouse button while still over the Cancel button. If so, it calls the CancelProcess() method to cancel the process. This is a roundabout way of doing it but is *much* faster than using DOEVENTS.

```
local llReturn
llReturn = .F.
with This
  if .cmdCancel.Visible and mdown() and ;
    .IsMouseOverCancel()
    doevents
    if not mdown() and .IsMouseOverCancel()
      llReturn = .T.
      .CancelProcess()
    endif not mdown() ...
  endif .cmdCancel.Visible ...
endwith
return llReturn
```

The protected method IsMouseOverCancel() returns .T. if the mouse is over the Cancel button. It uses the SYS(1270) function to get a reference to the object the mouse is over, and returns .T. if that object is the Cancel button.

```
local oControl
oControl = sys(1270)
return type('oControl') = 'O' and ;
```

```
  not isnull(oControl) and ;
  oControl.Name = 'cmdCancel' and ;
  oControl.Parent.Name = This.Name
```

CancelProcess() is a protected method that cancels the process by setting the variable or property specified in cCancelProperty to .T.:

```
with This
  if not empty(.cCancelProperty) and ;
    type(.cCancelProperty) = 'L'
    store .T. to (.cCancelProperty)
  endif not empty(.cCancelProperty) ...
endwith
```

Conclusion

I hope you like the new focus of this column. Having already tasted the benefits of this approach, I feel strongly that the future of software development is the assembly of reusable components whenever possible rather than hand-crafting code. Together, we'll build a toolbox of components that you can use in lots of applications. I'm always open to new ideas, so send me any suggestions for components you'd like to see.

*** NOTE: The following should go in a pullout box or something:

The April 1997 issue presented a program call XTABREP.PRG for creating horizontally paginated reports. While this program worked perfectly on my laptop, fellow FoxTalk author John Peterson found that it didn't output all of the columns properly on his system. The problem turned out to be related to printer driver settings. The Subscriber Downloads has an updated version of this program that eliminates printer driver dependencies.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Sask., Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America, and will be speaking at the 1997 Microsoft FoxPro Developers Conference. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326.