Raise a Little Event of Your Own

Doug Hennig

Event binding, added in VFP 8, is getting more and more used as VFP developers discover its powerful capabilities. This month, Doug Hennig examines a couple of uses he's been making of this technique lately.

I've been using event binding more and more in my applications these days. What is event binding and what's it good for? I first discussed one use for event binding in my June 2003 FoxTalk article, "Ahoy! Anchoring Made Easy." That article discussed the use of event binding to provide anchoring in controls. Of course, since VFP 9 now natively provides anchoring, I don't use that technique any more. However, I've recently used event binding to solve a couple of thorny problems I've had to deal with in the past in new elegant ways.

First, a brief primer on event binding.

Event binding in VFP

Windows fires events when things happen. For example, when the user clicks on a button, the Click event of that button fires. That event is handled by any code you place into the Click method of the button. (Here, I distinguish between events, which are fired by the system when something happens, and methods, which are automatically called when the event fires.) You could say that the Click method is bound to the Click event. However, this binding is within a single object. What if one object wants to receive notification when an event occurs in another object? That's what event binding is about.

VFP 7 added the ability to bind to events in COM objects with the EVENTHANDLER() function. (With earlier versions, you could do event binding using the separate VFPCOM utility.) This allows us to, for example, do something in VFP when a message is received in Outlook or a document is opened in Word. VFP 8 extended this ability by providing event binding to native controls. This is done with the BINDEVENT() function. Here's an example:

```
bindevent(Object1, 'Click', Object2, 'HandleClick')
```

This code sets up event binding between Object1 and Object2. In this case, when the Click event for Object1 fires, the HandleClick method of Object2 is automatically called.

Calling a method in code, such as SomeObject.SomeMethod(), doesn't fire a event, so anything bound to the SomeMethod method of SomeObject won't receive notification. If you want an event to fire and event binding to work, use the RAISEEVENT() function rather than calling SomeMethod. For example, use:

```
raiseevent(SomeObject, 'SomeMethod')
```

SomeMethod is called but so is the method of any object bound to the SomeMethod event. You can even pass parameters to these methods, such as:

```
raiseevent(SomeObject, 'SomeMethod', MyParameter1, MyParameter2)
```

Note that when you use RAISEEVENT(), any return value from either the event code or any methods bound to the event are ignored; RAISEEVENT() always returns .T.

Two other functions are related to event binding. As its name implies, UNBINDEVENTS() turns off event binding. AEVENTS() fills an array with information about event binding.

BINDEVENT(), UNBINDEVENTS(), and AEVENTS() have various parameters that specify exactly what they should do. I won't go into any more detail about event binding here; see the VFP help topics for these functions for more information.

Status messaging

One of the uses I've found for event binding is object messaging without requiring two-way object references. For example, say you have an object that performs some lengthy processing. It would be nice to show the user the status of the processing so they don't think the application is hung.

ProcessEngine, in SAMPLES.VCX, is a simple example of such a class. It has two methods, Process1 and Process2, that don't actually do anything; they just fake a multi-step process by pausing for a second between "tasks." Here's the code in Process1:

```
* Declare the Windows API Sleep function so we can
* pause for a while.

declare Sleep in Win32API integer nMilliseconds

* Perform each task and update the status.

This.oStatus.ShowStatus('Performing task 1')
Sleep(1000)
This.oStatus.ShowStatus('Performing task 2')
Sleep(1000)
This.oStatus.ShowStatus('Performing task 3')
Sleep(1000)
This.oStatus.ShowStatus('Done')
```

This class can be used from a variety of sources, such as a form. Here's an example, taken from the Click method of the command button in PROCESS.SCX:

```
loProcess = newobject('ProcessEngine', 'Samples.vcx')
loProcess.oStatus = Thisform
loProcess.Process1()
```

Process1 calls the ShowStatus method of the form (via its reference in oStatus), which updates an ActiveX StatusBar control to inform the user of the current status.

The problems with this code are that it assumes there's an object reference stored in the oStatus property, that oStatus has a ShowStatus method, and that only one object needs to be notified of changes. What if the developer forgets to put a reference to the form into the oStatus property? What if the calling form doesn't have a ShowStatus method? What if we want to build ProcessEngine into a COM object and call it from a Web service where there's no user interface? What if several objects need to know about status changes?

There's another issue as well: the client has a reference to the ProcessEngine object and the ProcessEngine object has a reference to the client (or at least some other object capable of displaying the current status). This can cause problems when it's time to destroy the objects: the client object can't be destroyed because there's an outstanding reference to it in ProcessEngine and the ProcessEngine object can't be destroyed because there's an outstanding reference to it in the client. This isn't an insurmountable problem—you need to set the outstanding references to NULL before destroying the objects—but one you need to be aware of.

In VFP 8 and later, we can take advantage of event binding to reduce the coupling. Here's the code in the Process2 method of ProcessEngine:

```
raiseevent(This, 'ShowStatus', 'Performing task 1')
Sleep(1000)
raiseevent(This, 'ShowStatus', 'Performing task 2')
Sleep(1000)
raiseevent(This, 'ShowStatus', 'Performing task 3')
Sleep(1000)
raiseevent(This, 'ShowStatus', 'Done')
```

Notice there's no reference to any object other than itself in this code. That means this object has no idea whether a client needs to be notified of status changes or not, nor does it care. There's no concern about oStatus containing a valid object reference and that object having a ShowStatus method. All that's required is that this class has a ShowStatus method that accepts a string parameter because of the way this

code calls RAISEEVENT(). The ShowStatus method in ProcessEngine contains a LPARAMETERS statement and nothing else, since no behavior is needed, just its existence.

Here's the code in the Click method of the command button in PROCESS.SCX that calls Process2:

```
loProcess = newobject('ProcessEngine', 'Samples.vcx')
bindevent(loProcess, 'ShowStatus', Thisform, ;
  'ShowStatus')
loProcess.Process2()
unbindevents(loProcess)
```

By binding to the ShowStatus event of the process class, the form gets notification of status changes without the coupling between the objects we saw earlier. If the form's method is called UpdateStatus instead of ShowStatus, we just have to change the BINDEVENT() statement and not touch the ProcessEngine class at all. If we want to use ProcessEngine from an object that doesn't require any user interface, we simply don't use BINDEVENT() and no status update messaging occurs. If we want more than one object to be notified about the status changes, we just add more BINDEVENT() statements.

The cool thing about this is that there isn't a lot of reengineering required to implement it:

- Add a method to the "server" class that acts as the "event." No code (other than an LPARAMETERS statement if any parameters are passed) is required, although there could be if you want some behavior to occur.
- In server class methods, change the calls to the other object ("This.oStatus.ShowStatus" in this case) to RAISEEVENT().
- In the "client" object, don't store an object reference in a property of the server object ("oStatus" in this example), and use BINDEVENT() and UNBINDEVENTS() to bind the server event to the object's method.

In less than an hour, I refactored an entire application to use this form of messaging rather than the more coupled form. As a result, my classes are much more flexible that before.

Change messaging

VFP guru Tamar Granor and I were chatting one day about the best way to let objects in a form know when the user changes something. This is typically used to enable Save and Cancel buttons in a data entry form, or the Next and Finish buttons in a wizard-based form, or other related controls in any type of form.

I've used a variety of mechanisms over the years to handle this situation.

Controls call a form method

The InteractiveChange and ProgrammaticChange methods of every control in a form call a custom RefreshOnChange method of the form (the InteractiveChange and ProgrammaticChange of my base classes, such as SFTextBox and SFEditBox, call a custom AnyChange method of the class, so that's typically where I'd put the code). RefreshOnChange then enables or disables controls, such as Save and Cancel buttons, as necessary.

There are several problems with this method. First, I had to put the call into AnyChange, which I typically did at the class level to save doing it manually in every control, which meant creating an entirely new later of classes and then being limited to using those classes. Second, each form had to know which controls to refresh and their hierarchy (what page in a pageframe, for example, or in the associated toolbar). As a workaround for the second issue, the Refresh method of those controls that care when things change can enable or disable the controls so the form doesn't have to, but that means refreshing the entire form, which then has consequences for the control that has focus.

A timer calls a form method

The Timer method of a timer on a form calls the IsRecordChanged method of the form to determine, using GETFLDSTATE(), if anything in the current record the controls are bound to has changed, and if so, calls the RefreshOnChange method of the form, which deals with notifying the other controls using similar code to the previous mechanism.

This mechanism resolves the first problem described above but still suffers from the second, and has a new problem that the controls are only refreshed as often as the timer fires. Also, this only works if the controls are bound to fields in a cursor; it doesn't work if they're bound to properties of an object, such as a business object, unless there's a means to determine when those properties have changed.

Only refreshing objects that need refreshing

The classes I presented in my May 1998 article, "Build Your Own Wizards," for creating wizard-based forms have a custom lRefreshSteps property. If this property is .T., the control should be refreshed when something in the form changes. The InteractiveChange and ProgrammaticChange methods of these controls call the RefreshSteps method of the form, which iterates through all the controls in the form, calling the Refresh method of each one that has lRefreshSteps set to .T. The Refresh method of those controls then take the appropriate action.

This solution eliminates the second problem (the form only refreshes those controls that indicate they need refreshing) but still suffers from the first, although since the wizard-based controls have other wizard-related behavior and would normally be used anyway, this isn't as big a problem.

Using event binding

Tamar and I came to the realization that event binding would provide a neat solution to these issues. Here's how I implemented it.

As I mentioned earlier, the InteractiveChange and ProgrammaticChange of my base classes call the custom AnyChange method. In the new iteration, these methods do this instead:

```
raiseevent(This, 'AnyChange')
```

From the point of view of the object, the effect is the same: the AnyChange method is called. However, the difference is that other objects can now bind to AnyChange and be notified of the change without the object being aware of this need.

The Init method of these classes has the following code:

```
if This.lBindToFormAnyChange and ;
  vartype(Thisform) = 'O' and ;
  pemstatus(Thisform, 'AnyChange', 5)
  bindevent(This, 'AnyChange', Thisform, 'AnyChange')
endif This.lBindToFormAnyChange ...
```

If the custom lBindToFormAnyChange property is .T. (it's .F. by default), this code binds the AnyChange method of the object to the AnyChange method of the form (if such a method exists). SFForm, my base class Form object, does have any AnyChange method. This method simply sets the custom lChanged property of the form to .T. to indicate that something changed.

All controls have a custom lNotifyOnFormChange property that, if .T. (it's .F. by default), tells the Init method to bind the control's OnFormChanged method to the lChanged property of the form (yes, you can bind to a property, which causes the bound event of the object to fire whenever the property's value changes). Notice the additional "1" specified in BINDEVENT(); this tells VFP to call OnFormChanged after lChanged is changed rather than before. This is important because the control may do something with lChanged, and we want it to see the new value rather than the former one.

```
if This.lNotifyOnFormChange and ;
  vartype(Thisform) = 'O' and ;
  pemstatus(Thisform, 'lChanged', 5)
  bindevent(Thisform, 'lChanged', This, ;
    'OnFormChange', 1)
endif This.lNotifyOnFormChange ...
```

By default, OnFormChanged calls This.Refresh, but in a subclass or instance, you could have it do something different if necessary.

So, here's how this all works. I add a control to the form. If the form or other controls need to know when the value of the control changes, I set its lBindToFormAnyChange property to .T. If the control needs

to know when something else in the form changes, I set its lNotifyOnFormChange property to .T. and put the appropriate code into its Refresh or OnFormChanged method.

For example, in CUSTOMERS.SCX, the lBindToFormAnyChange property of every text box is set to .T., so changes in these controls causes the form to be notified, and the lNotifyOnFormChange property of the Save and Cancel buttons is set to .T., so they're notified when the form is. The following code in the Refresh method of these buttons enables them when the user edits the current record:

```
This.Enabled = Thisform.lChanged
```

Better change control

What if a user changes the value in a textbox, then changes the value back? We don't really need to enable the Save and Cancel buttons in that case. The problem is that every change causes the form's lChanged property to be set to .T., even if the change undoes a previous change. What we really need to do is check whether there are any changes in the underlying data, and if so, set lChanged appropriately.

SFMaintForm, my data entry form base class, overrides the AnyChange method to check if the control that caused the method to be fired (which we can get from the AEVENTS() function) is bound to a field in a cursor, and if so, if the value of that field has changed. This codes calls a custom UpdateChanges method, which we won't look at here, to update a collection containing the names of the fields that have changed. If a field is changed, UpdateChanges adds it to the collection. If the field is then changed back to its original value, UpdateChanges removes it from the collection. If the collection is empty, there are no changes, so lChanged is set to .F. Here's the code for AnyChanges:

```
local laEvent[1], ;
  loControl, ;
  lcField, ;
  lnPos, ;
  lcAlias, ;
  luValue, ;
  luOldValue, ;
  llChanged
aevents(laEvent, 0)
loControl = laEvent[1]
do case

* If the control that triggered AnyChange has a
* ControlSource property and it points to something,
* figure out what.

  case pemstatus(loControl, 'ControlSource', 5) and ;
    not empty(loControl.ControlSource)
    lcField = loControl.ControlSource
    lnPos   = at('.', lcField)
    lcAlias = left(lcField, lnPos - 1)
    lcField = substr(lcField, lnPos + 1)

* If the control has a Value property, get the value.

    if pemstatus(loControl, 'Value', 5)
      luValue = loControl.Value

* If the control is bound to a field in a cursor, see
* if the value is different from the field. We could
* use GETFLDSTATE(), but it indicates that a field
* changed even if the field was replaced with the
* same value. So, we'll compare the current value
* against OLDVAL() instead.

      if used(lcAlias) or empty(lcAlias)
        luOldValue = oldval(lcField, lcAlias)
        llChanged  = (isnull(luOldValue) and ;
          not empty(luValue)) or ;
          (not luOldValue == luValue)

* We could use code like "This.lChanged =
```

```
* This.lChanged OR llChanged", but once This.lChanged
* is .T., reverting a field to its former value
* leaves This.lChanged at .T. because of the OR.
* Instead, we'll fill a collection with the names of
* changed fields. If the field is reverted, we remove
* it from the collection. Thus, This.lChanged is only
* .T. if there are any items in the collection. This
* work is done in UpdateChanges.

        This.lChanged = This.UpdateChanges(lcAlias ;
          + '.' + lcField, llChanged)

* The control is bound to something else, so just
* flag that something has changed.

      else
        This.lChanged = .T.
      endif used(lcAlias) ...
    endif pemstatus(loControl, 'Value', 5)

* Other cases could go here, such as a custom
* cControlSource property. If we don't know how to
* handle it specifically, just flag that something
* has changed.

  otherwise
    dodefault()
endcase
```

Run CUSTOMERS.SCX and make some changes to some of the fields. Notice that the Save and Cancel buttons become enabled as soon as you make the first change. Then undo the changes you made and notice that these buttons become disabled again.

Summary

Event binding is a wonderful addition to VFP because it allows you to hook objects together without them knowing about it or having the same coupling required without this technique. I'm sure I'll find more uses for it as times goes on.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com