

The Straight POOP: Event binding and loose coupling

Nancy Folsom

This month I take a look at a feature that is new to VFP 8.0 and relate it to the object oriented programming concept of loose coupling.

Sorry, but I'm afraid this is G-rated, so this article won't be like *that*. In this context, coupling refers to the degree of dependency between objects. If there is little dependency, then a system is *loosely coupled*. If objects refer directly to other objects, then they are *tightly coupled*. There has always been at least some dependency between containers and their contained objects, although it's been possible to eliminate most if not all dependency between the objects within a container, be it a control class, container, form, and so on. Why? One of the first questions most people run into when starting to use Visual FoxPro is the problem of what happens when one renames, or even deletes, an object in a container that other objects refer to. When objects are tightly coupled, it's hard to change their behavior, swap them out for other objects, and trace bugs, since it can be a struggle to find where in the object hierarchy the problem is hiding.

Where does coupling come-in?

Most of us create detailed data entry forms that do a number of common tasks. Users can input data, change data, save, or undo edits, and they might be able to lock a record against editing. Figure 1 shows a simple example of a data entry form.

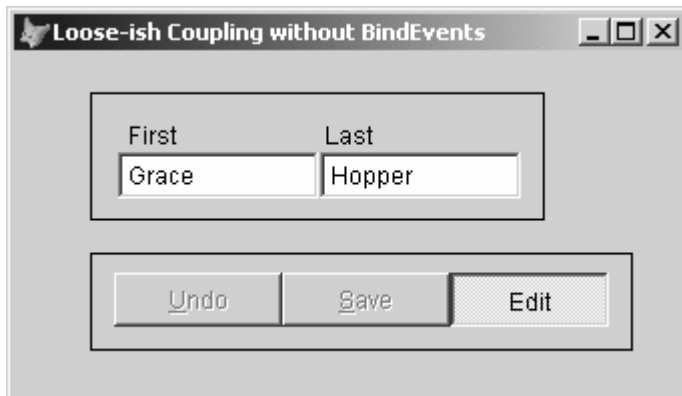


Figure 1 State of UI when a form is first opened (SPOOP9Fig1.tif) in edit mode

In order to help users make sensible choices, it's helpful to disable the buttons that don't make sense in a given context. In the example, once the data is changed, the Save and Undo buttons are enabled, signaling to the user that there are pending changes. Figure 2 shows this state. If I click on the Edit button to lock the data from accidental editing, the data fields are disabled, as Figure 3 shows.

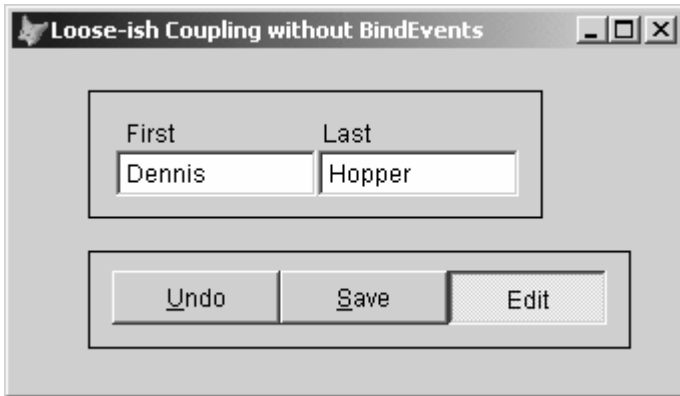


Figure 2 State of UI elements when data has changed



Figure 3 View of UI after edit mode is turned off

Coordinating members

Coordinating various elements on the form that have to interact, and, yet, are independent components can be tricky to implement so that the dependencies are minimal. In the example, the container that has the data is one logical unit, and the container of buttons is another. The buttons should be reusable on any data entry form, and the data will exist independently of what actions might be available. There is a third unit formed by the *relationship* between the button container and the data container.

The relationship between the data and the available functions (save, undo, edit) is represented by the form, in this example. The form will coordinate the two containers. The less the two containers know about each other, the better for reusability. One might want to save (or undo) changes to not just person-related data, but also to car models, inventory items, and so on. The container of person-related data might be reused on a page in a page frame, displayed as read-only for reporting, and so on. Each class has some fundamental responsibilities. The button container has to be able to enable and disable buttons, and it has to have some method that parallels the functionality represented by the buttons. In other words, the button container will have a method for each of the button's functions that the buttons can call when they're clicked.

Why not put the functionality *in* the button? Arguably, the first step to achieving the enlightened state of loose coupling is to eliminate references that objects make to the objects contained *within* another container. So, for example, eliminate references like the following:

```
*** SomeForm.SomePageFrame.SomePage.SomeButton.Click()
THIS.PARENT.Page2.Text1.VALUE = "I've changed!"
```

Instead, let the containers simply notify the mediator (the form) that something has occurred, but then leave it up to the form to do something with the information. Listing 1 shows a simplistic way of decoupling the button and data containers on a data entry form. In this scenario, a form has two containers: one displays data from a record, and the other container has 3 buttons for Save, Undo, and Edit Mode. When data is

changed, the Save and Undo buttons are enabled. When the Edit Mode is False, the data objects are disabled. When changes are saved or reversed, the Save and Undo buttons need to be disabled and the data needs to be saved, or reverted. When the Edit Mode is turned off, any pending changes are saved. So, in short, the buttons and the data objects have effects beyond their immediate responsibilities. In order to accomplish this, the buttons pass their messages up to their owner-container, which passes the messages up to the form, which then passes the messages down to the data container, which, finally, does something (or not) with the data objects. Somewhat ironically, in order to decouple the logic in a Lastname TextBox from the logic in a Save button, many objects need to be involved. It's only an apparent irony, though. Any tight coupling occurs between an object and its parent, which is acceptable so long as the container is solely responsible for communicating with the world. However, there's plenty of snug-coupling, though.

The code listing shows that containers have methods that parallel the events they'll mediate. When the Save button is clicked, it calls the button container's Save method, which calls the mediator's Save(). The mediator (the form) calls the data container's Save(). This what I mean when I say snug-coupling. Although the button container and the data container in this example are decoupled, there is still more dependency between the form and the containers than is comfortable. First, it's difficult to get the synchronization correct (it's best to implement one bit of functionality at a time and test it), second, it's difficult to remember where in the hierarchy one put the critical code, and, third, if you want to drop the button container on a different form, for example, you have to make sure that the mediator methods (such as Save) are present.

Event binding

Happily, Visual FoxPro 8.0 allows us to decouple complex messaging logic like in my previous example, by allowing us to raise, bind to, and chain events together. We can even treat events like objects. First, let me back up a quick moment. Events are different from methods in this way: events occur automatically, under certain circumstances. Methods run only when we invoke them programmatically, for example. One can still call events as one would invoke methods, but it's better to not.

Events should only run when VFP thinks they should run. So, instead of calling a button's click, for example, it's better to have a form-level method called, say, OnClick() that both your code elsewhere and the button click() event can call. In the previous example, the save and undo buttons call method code in the container that saves, or reverses, changes, respectively.

Event *binding* means that we can associate the events that occur in one object with events in another object. Events do not need to be named the same in both objects. There are some complexities to the syntax and use, which are not the subject of this particular article. Please see Mike Helland's introduction to Event Binding in last month's issue. What is important to consider with respect to coupling, is that my mediator can now replace all its custom methods for Save, Undo, and so on, and the containers can ignore notifying a parent about events. It's up to the mediator to set up the event binding between objects. Let's get to an example. Listing 2 rewrites the first example to make use of the BindEvent() function, new to VFP 8.0. The critical differences between the two methodologies is that containers don't have parallel methods that contained objects call when their state changes, and there is less need for Access and Assign methods. Here are some of the relevant code snippets from the listing.

```
LOCAL loForm AS FORM
loForm = NEWOBJECT("BindEvents")
loForm.SHOW(1)

DEFINE CLASS BindEvents AS FORM
...

PROCEDURE INIT
    ** Set up event bindings.

    ** When the edit button is clicked, trigger the data container's enable method.
    BINDEVENT(THIS.objDataCmd,"SetEdit",THIS.objPerson,"EnableControls",2)

    ** If we're switching out of edit mode, save any changes.
    BINDEVENT(THIS.objDataCmd,"SetEdit",THIS.objDataCmd,"Save",2)

    ** When data is being changed, trigger the button container refresh.
    BINDEVENT(THIS.objPerson,"OnChange",THIS.objDataCmd,"OnChange",2)
```

```

    !* When Save is clicked, trigger the data container to save.
    BINDEVENT(THIS.objDataCmd,"Save",THIS.objPerson,"Save",2)

    !* When Undo is clicked, trigger the data
    !* container to revert changes.
    BINDEVENT(THIS.objDataCmd,"Undo",;
      THIS.objPerson,"Undo",2)
  ENDPROC

```

ENDDDEFINE

While the form still has two containers: one with buttons and one with data objects, the form no longer needs to have matching, parallel methods (like Save). Instead the Form sets up the relationship between the data and buttons by relating the two relevant methods in the containers. Notice, too, that events can be bound to more than one event. In the above example, the SetEdit will both Save and EnableControls. This is at the highest level. Even within the containers, event binding simplifies the task of objects communicating with their parent.

I added a custom method to the container of buttons for Save, Undo, and Edit Mode called BindEvents(), which I call from the Init(). This method lets the container hook into the interesting events of the buttons. In this case the container is notified when the edit mode changes, and when either Save or Undo is clicked. The Save and Undo click events don't even have any code in them.

```

DEFINE CLASS DataCommands AS CONTAINER
...
  PROCEDURE BindEvents
    !* Instead of button's click() calling a
    !* method in the parent container. Simply
    !* use the edit button's own event to trigger
    !* the parent to some action.
    BINDEVENT(THIS.btnEdit,"InteractiveChange",;
      THIS,"SetEdit",2)
    BINDEVENT(THIS.btnEdit,"ProgrammaticChange",;
      THIS,"SetEdit",2)
    BINDEVENT(THIS.btnUndo,"Click",THIS,"Undo",2)
    BINDEVENT(THIS.btnSave,"Click",THIS,"Save",2)
  ENDPROC

```

I do something similar with the data container. I bind a customer container method (OnChange) to the InteractiveChange() events of each TextBox. So, instead of calling the EntityContainer.OnChange custom method from the TextBox InteractiveChange(), the EntityContainer defines, once, that InteractiveChange() events will run not only any code that might be in them, but also the OnChange method. In this case, the OnChange method doesn't do anything, however the form can bind this method to the buttons container method, called OnChange, coincidentally, that enables the Save and Undo buttons when there are changes pending.

Why is this a good thing?

BindEvent() takes us one step closer to loosely coupled implementations. Classes can be written to manage their own unit of work, without having to be designed to pass actions and messages to a parent, which then passes on the action or message, to other objects. The same functionality is achieved by simply *binding* the events together within the context they cooperate, such as in a form. In addition, this means that even our runtime code can, on-the-fly, create event bindings for a dynamic system. And event handlers can be objectified and thus attached to objects just as we're getting used to doing with business rules. Visual FoxPro 7.0 and now 8.0 offer ever more ways to implement object orientation in our applications.

Program listings

Listing 1

```
Local loForm As Form
loForm = NewObject("NoBindEvents")
loForm.Show(1)

Define Class NoBindEvents As Form

    Height = 170
    Width = 334
    Caption = "Loose-ish Coupling without BindEvents"
    EditMode = .T.
    DirtyBuffer = .F.
    Name = "NoBindEvents"

    !* Container responsible for displaying data
    Add Object ObjPerson As EntityContainer With ;
        TOP = 20, ;
        LEFT = 38, ;
        NAME = "objPerson", ;
        Label1.Name = "Label1", ;
        Label2.Name = "Label2", ;
        txtFirst.Name = "txtFirst", ;
        txtLast.Name = "txtLast"

    !* Container responsible for managing buttons that can
    !* trigger actions
    Add Object ObjDataCmd As DataCommands With ;
        TOP = 100, ;
        LEFT = 38, ;
        NAME = "objDataCmd", ;
        btnEdit.Name = "btnEdit", ;
        btnUndo.Name = "btnUndo", ;
        btnSave.Name = "btnSave"

    !* When the edit mode changes, alert the data container
    Procedure EditMode_Assign
        Lparameters vNewVal
        This.EditMode = m.vNewVal
        This.ObjPerson.EnableControls(m.vNewVal)
    Endproc

    !* Method parallels container actions. Used for mediation.
    Procedure Save
        This.ObjPerson.Save()
    Endproc

    !* Method parallels container actions. Used for mediation.
    Procedure Undo
        This.ObjPerson.Undo()
    Endproc

    Procedure DirtyBuffer_Assign
        Lparameters vNewVal
        This.ObjDataCmd.OnChange()
    Endproc

    !* One of the significant (public) events is when data changes.
    Procedure ObjPerson.DirtyBuffer_Assign
        Lparameters vNewVal
        Store m.vNewVal To ;
            THIS.DirtyBuffer, ;
            THIS.Parent.DirtyBuffer
    Endproc

    !* One of the significant (public) events is when the edit mode changes.
    Procedure ObjDataCmd.EditMode_Assign
        Lparameters vNewVal
        Store vNewVal To This.EditMode, This.Parent.EditMode
```

```

Endproc

** When Undo is selected, the button container first tells the
** mediator (form), so it can do whatever it needs to, and then
** the container takes care of its internal business. In this case,
** the container resets the buttons' enabled property.
Procedure ObjDataCmd.Undo
    This.Parent.Undo()
    DoDefault()
Endproc

** When Save is selected, the button container first tells the
** mediator (form), so it can do whatever it needs to, and then
** the container takes care of its internal business. In this case,
** the container resets the buttons' enabled property.
Procedure ObjDataCmd.Save
    This.Parent.Save()
    DoDefault()
Endproc

Enddefine

** Container of Save, Undo, and Edit buttons...like a CommandGroup
Define Class DataCommands As Container

    Width = 271
    Height = 49
    EditMode = .T.
    Name = "DataCommands"
    DirtyBuffer = .F.

    ** Uncheck Edit to lock the data against edits
    Add Object btnEdit As Checkbox With ;
        TOP = 10, ;
        LEFT = 180, ;
        HEIGHT = 27, ;
        WIDTH = 79, ;
        CAPTION = "\<Edit", ;
        VALUE = .T., ;
        CONTROLSOURCE = "THIS.PARENT.EditMode", ;
        STYLE = 1, ;
        NAME = "btnEdit"

    ** Button will undo changes since the last save
    Add Object btnUndo As CommandButton With ;
        TOP = 10, ;
        LEFT = 12, ;
        HEIGHT = 27, ;
        WIDTH = 84, ;
        CAPTION = "\<Undo", ;
        ENABLED = .F., ;
        NAME = "btnUndo"

    ** Save pending changes
    Add Object btnSave As CommandButton With ;
        TOP = 10, ;
        LEFT = 96, ;
        HEIGHT = 27, ;
        WIDTH = 84, ;
        CAPTION = "\<Save", ;
        ENABLED = .F., ;
        NAME = "btnSave"

    ** Provide Assign methods to properties that
    ** represent significant (i.e. public) events. This is
    ** helpful for leaving a hook for a container to tell a
    ** mediator that something has happened.
    Procedure EditMode_Assign
        lparameters vNewVal
        This.EditMode = m.vNewVal
    Endproc

```

```

** Dirty buffer is a logical property reflecting whether
** there are any changes to the data.
Procedure DirtyBuffer_Assign
  Lparameters vNewVal
  This.DirtyBuffer = m.vNewVal
Endproc

** Only enable save and undo buttons if there are changes
** to save or undo.
Procedure OnChange
  Store .T. To ;
  THIS.btnUndo.Enabled, ;
  THIS.btnSave.Enabled
Endproc

** Set the edit mode of the data (lock data against edits)
Procedure setedit
  Lparameters t1EditMode
  This.EditMode = t1EditMode
Endproc

** In real life, the container might notify the
** business object to save.
Procedure Save
  Store .F. To This.DirtyBuffer
  This.OnSave()
Endproc

** In real life, the container might notify the
** business object to undo changes.
Procedure Undo
  Store .F. To This.DirtyBuffer
  This.OnSave()
Endproc

** Once changes are saved or reversed, there aren't any more
** pending changes, so disable these buttons.
Procedure OnSave
  Store .F. To This.btnUndo.Enabled, This.btnSave.Enabled
Endproc

** Container buttons simply call the container's parallel method
Procedure btnUndo.Click
  This.Parent.Undo()
Endproc
Procedure btnSave.Click
  This.Parent.Save()
Endproc

Enddefine

** Container of data objects, for editing, viewing, and so on.
Define Class EntityContainer As Container

  Width = 227
  Height = 64
  Name = "EntityContainer"
  DirtyBuffer = .F.

  ** The usual textboxes and labels for displaying or editing
  ** a first name and a last name.
  Add Object Labell As Label With ;
  BACKSTYLE = 0, ;
  CAPTION = "First", ;
  HEIGHT = 17, ;
  LEFT = 19, ;
  TOP = 14, ;
  WIDTH = 40, ;
  NAME = "Labell"

```

```

Add Object Label2 As Label With ;
  BACKSTYLE = 0, ;
  CAPTION = "Last", ;
  HEIGHT = 17, ;
  LEFT = 117, ;
  TOP = 14, ;
  WIDTH = 40, ;
  NAME = "Label2"

Add Object txtFirst As TextBox With ;
  HEIGHT = 23, ;
  LEFT = 14, ;
  TOP = 30, ;
  WIDTH = 100, ;
  VALUE = "Grace", ;
  NAME = "txtFirst"

Add Object txtLast As TextBox With ;
  HEIGHT = 23, ;
  LEFT = 115, ;
  TOP = 30, ;
  WIDTH = 100, ;
  VALUE = "Hopper", ;
  NAME = "txtLast"

Procedure EnableControls
  Lparameters tlEnable
  Store tlEnable To ;
    THIS.txtFirst.Enabled, ;
    THIS.txtLast.Enabled
Endproc

!* Normally a save would result in data changing.
Procedure Save
  Local loi As Object
  For Each loi In This.Controls
    If PEMSTATUS(loi, 'OldVal', 5)
      loi.Oldval = loi.Value
    Endif
  Next loi
Endproc

Procedure Undo
  Local loi As Object
  For Each loi In This.Controls
    If PEMSTATUS(loi, 'OldVal', 5)
      loi.Value = loi.Oldval
    Endif
  Next loi
Endproc

Procedure DirtyBuffer_Assign
  Lparameters vNewVal
  This.DirtyBuffer = m.vNewVal
Endproc

Procedure OnChange
Endproc

Procedure txtFirst.Init
  This.AddProperty('OldVal', This.Value)
Endproc

Procedure txtFirst.GotFocus
  This.Oldval = This.Value
Endproc

!* Interactive change is important trigger for starting the process
!* of alerting all who might care that data has changed.
Procedure txtFirst.InteractiveChange
  This.Parent.DirtyBuffer = This.Oldval <> This.Value

```

```
Endproc

Procedure txtLast.GotFocus
  This.Oldval = This.Value
Endproc

Procedure txtLast.Init
  This.AddProperty('OldVal',This.Value)
Endproc

Procedure txtLast.InteractiveChange
  This.Parent.DirtyBuffer = This.Oldval <> This.Value
Endproc

Enddefine
```

Listing 2

```
Local loForm As Form
loForm = Newobject("BindEvents")
loForm.Show(1)

Define Class BindEvents As Form

    Height = 170
    Width = 334
    Caption = "Loose Coupling with BindEvents"
    Name = "BindEvents"

    !* Container responsible for displaying data
    Add Object objPerson As EntityContainer With ;
        TOP = 20, ;
        LEFT = 38, ;
        NAME = "objPerson", ;
        Label1.Name = "Label1", ;
        Label2.Name = "Label2", ;
        txtFirst.Name = "txtFirst", ;
        txtLast.Name = "txtLast"

    !* Container responsible for managing buttons that can
    !* trigger actions
    Add Object objDataCmd As DataCommands With ;
        TOP = 100, ;
        LEFT = 38, ;
        NAME = "ObjDataCmd", ;
        btnEdit.Name = "btnEdit", ;
        btnUndo.Name = "btnUndo", ;
        btnSave.Name = "btnSave"

    Procedure Init
        !* Set up event bindings.

        !* When the edit button is clicked, trigger the data container's enable method.
        Bindevent(This.objDataCmd,"SetEdit",This.objPerson,"EnableControls",2)

        !* When data is being changed, trigger the button container refresh.
        Bindevent(This.objPerson,"OnChange",This.objDataCmd,"OnChange",2)

        !* When Save is clicked, trigger the data container to save.
        Bindevent(This.objDataCmd,"Save",This.objPerson,"Save",2)

        !* When Undo is clicked, trigger the data container to revert changes.
        Bindevent(This.objDataCmd,"Undo",This.objPerson,"Undo",2)
    Endproc
Enddefine

Define Class DataCommands As Container

    Width = 271
    Height = 49
    Name = "DataCommands"
    DirtyBuffer = .F.

    Add Object btnEdit As Checkbox With ;
        TOP = 10, ;
        LEFT = 180, ;
        HEIGHT = 27, ;
        WIDTH = 79, ;
        CAPTION = "\<Edit", ;
        VALUE = .T., ;
        STYLE = 1, ;
        NAME = "btnEdit"
```

```

Add Object btnUndo As CommandButton With ;
    TOP = 10, ;
    LEFT = 12, ;
    HEIGHT = 27, ;
    WIDTH = 84, ;
    CAPTION = "\<Undo", ;
    ENABLED = .F., ;
    NAME = "btnUndo"

Add Object btnSave As CommandButton With ;
    TOP = 10, ;
    LEFT = 96, ;
    HEIGHT = 27, ;
    WIDTH = 84, ;
    CAPTION = "\<Save", ;
    ENABLED = .F., ;
    NAME = "btnSave"

Procedure BindEvents
    !* Instead of button's click() calling a method in the parent container,
    !* Simply use the edit button's own event to trigger the parent to some action.
    Bindevent(This.btnEdit,"InteractiveChange",This,"SetEdit",2)
    Bindevent(This.btnEdit,"ProgrammaticChange",This,"SetEdit",2)
    Bindevent(This.btnUndo,"Click",This,"Undo",2)
    Bindevent(This.btnSave,"Click",This,"Save",2)
Endproc

!* Method that can be called when data is changed.
Procedure OnChange
    If .Not. This.DirtyBuffer
        Store .T. To ;
            THIS.DirtyBuffer, ;
            THIS.btnUndo.Enabled, ;
            THIS.btnSave.Enabled
    Endif
Endproc

!* Method that can be called when data is saved.
Procedure Save
    Store .F. To This.btnSave.Enabled, ;
        THIS.btnUndo.Enabled, ;
        THIS.DirtyBuffer
Endproc

!* Method that can be called when data is reverted.
Procedure Undo
    Store .F. To This.btnSave.Enabled, ;
        THIS.btnUndo.Enabled, ;
        THIS.DirtyBuffer
Endproc

Procedure Init
    This.BindEvents()
Endproc

Procedure SetEdit
Endproc

Enddefine

Define Class EntityContainer As Container

    Width = 227
    Height = 64
    Name = "EntityContainer "

```

```

Add Object Label1 As Label With ;
  BACKSTYLE = 0, ;
  CAPTION = "First", ;
  HEIGHT = 17, ;
  LEFT = 19, ;
  TOP = 14, ;
  WIDTH = 40, ;
  NAME = "Label1"

Add Object Label2 As Label With ;
  BACKSTYLE = 0, ;
  CAPTION = "Last", ;
  HEIGHT = 17, ;
  LEFT = 117, ;
  TOP = 14, ;
  WIDTH = 40, ;
  NAME = "Label2"

Add Object txtFirst As TextBox With ;
  HEIGHT = 23, ;
  LEFT = 14, ;
  TOP = 30, ;
  WIDTH = 100, ;
  VALUE = "Grace", ;
  NAME = "txtFirst"

Add Object txtLast As TextBox With ;
  HEIGHT = 23, ;
  LEFT = 115, ;
  TOP = 30, ;
  WIDTH = 100, ;
  VALUE = "Hopper", ;
  NAME = "txtLast"

Procedure EnableControls
  This.txtFirst.Enabled = !This.txtFirst.Enabled
  This.txtLast.Enabled = !This.txtLast.Enabled
Endproc

Procedure BindEvents
  !** When data is changed, alert the parent container.
  Bindevent(This.txtFirst, "InteractiveChange", This, "OnChange", 3)
  Bindevent(This.txtLast, "InteractiveChange", This, "OnChange", 3)
Endproc

Procedure Save
  Local loi As Object
  For Each loi In This.Controls
    If PEMSTATUS(loi, 'OldVal', 5)
      loi.Oldval = loi.Value
    Endif
  Next loi
Endproc

Procedure Undo
  Local loi As Object
  For Each loi In This.Controls
    If PEMSTATUS(loi, 'OldVal', 5)
      loi.Value = loi.Oldval
    Endif
  Next loi
Endproc

Procedure Init
  This.BindEvents()
Endproc

Procedure OnChange
Endproc

Procedure txtFirst.InteractiveChange

```

```
Endproc

Procedure txtFirst.GotFocus
  This.Oldval = This.Value
Endproc

Procedure txtFirst.Init
  This.AddProperty('OldVal',This.Value)
Endproc

Procedure txtLast.Init
  This.AddProperty('OldVal',This.Value)
Endproc

Procedure txtLast.GotFocus
  This.Oldval = This.Value
Endproc

Enddefine
```

© 2002 Nancy Folsom, Pixel Dust Industries