August, 2005

Visual FoxPro 9/8

Fix Control Names Easily

This tool lets you specify a new name for each control on a form or class and fixes all code that uses it.

By Tamar E. Granor, technical editor

I'm currently working on an application that was originally written by someone else. It has several dozen forms and many of the forms have 15, 20, or more controls. The original author renamed the controls only sporadically, so most of them have names like Text1, Command2 (or even Command22), and so forth.

As I modify and improve these forms, figuring out which control I need to address is a real problem. I started renaming the controls manually, but making the corresponding changes to the form's code turned out to be extremely tedious and somewhat error-prone. (You have to be sure to rename them in the right order, as a search for Command2 also turns up Command20, Command21, and so forth). After I'd done just a few of them, it became clear I needed a tool to help.

I asked around, but no one knew of anything that would handle this task, so I decided to build my own. What I wanted was a tool that would:

- Audit a form to create a list of controls on the form
- Present me with that list and let me specify a new name for any of those controls
- Update both the code and the controls themselves to use the new names

It turned out the first two items on the list were easy, but the third offered some fairly interesting challenges. The biggest issue was properly identifying references to the different controls in the code -- that is, reading a line of code that works with a control and knowing which control on the form is being referenced, so the reference can be changed. Doing things in the right order turned out to be very important. In addition, along the way, I found two VFP bugs, one of which I have yet to work around. I call the resulting tool the Control Renamer.

In this article, I'll explain how it works and take a look at much of the code involved. After that, I'll show you how to use the tool.

Although I designed the tool to work with forms, it took only minor changes to get it to work with container classes as well. As I explain how it works, I'll generally refer to "the target object," which is the form or class you want to change. The Control Renamer is included on this issue's Professional Resource CD.

Structuring the tool

I decided early on to separate the engine for my tool from its user interface. This makes it possible to use the engine without a user interface, or to design a different user interface. The part of the tool that accomplishes the first and third tasks is handled by a subclass of Custom, while the user interface that lets you specify the new names is an SCX-based form.

After some consideration, I made the form the control center of this tool. So the form instantiates the engine class and maintains a reference to it. To use it, you open the target object in the Form Designer or Class Designer. Then you run the Control Renamer form either on its own or through VFP's Builder mechanism. (I show you later in this article how to hook the Control Renamer into the Builder system.)

The Control Renamer form

The Control Renamer form (figure 1) is fairly simple. It uses a list box to show the current names of the controls on the target object, has a few text boxes and an edit box to tell you more about the target object and the control currently selected in the list, and includes a button to start the renaming process. It also lets you decide whether to show the list of controls in alphabetical order or in container order. Container order -- the default -- means the controls inside a container are shown immediately after the container. Within any container (including the target object), controls are arranged alphabetically.

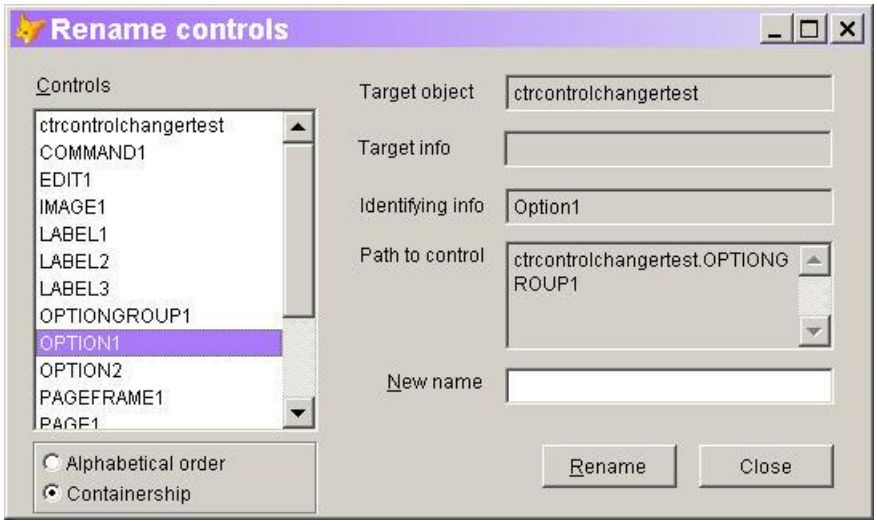


Figure 1: Control Renamer form–This shows you the controls on the target object and lets you specify new names. It's the control center for the tool.

The form has four custom properties and two custom methods (table 1). The About method is used only for documentation and isn't discussed further.

Table 1: Form PEMs–The Control Renamer form uses four custom properties and two custom methods.

Name	**Type**	**Purpose**
About	Method	Documentation
cObjectInfo	Property	Identifying information (such as caption) for the target object
FocusOnControl	Method	Moves focus on the target object to the control currently selected in the list box
oChangeForm	Property	Object reference to the Form Designer or Class Designer form for the target object
oChangeObject	Property	Object reference to the target object
oChangerEngine	Property	Object reference to the Control Renamer engine object

oChangeForm and oChangeObject make it possible to talk to the target object. oChangeObject is a reference to the form or class itself. It

turns out the Class Designer always creates a form, even when the class you're working with is based on some other base class. oChangeForm is an object reference to that form; it's used by the FocusOnControl method.

Managing focus in the target object

As I began to get the Control Renamer form working, I realized the value of synching the highlight in the target object with the list on the Control Renamer form. That is, as you move through the list, the appropriate control is highlighted on the target object. The FocusOnControl method accomplishes this.

Surprisingly, the SetFocus method works at design time, so FocusOnControl can call the SetFocus method of a control on the target object to highlight it in the designer. However, not every control in VFP has a SetFocus method. For those controls that don't, you have to use brute force. The code uses the MOUSE command to click on the appropriate control and, because it's rude for code to leave the mouse somewhere other than where it found it, MOUSE is used again to restore the mouse position. The Control Renamer form's AlwaysOnTop property is normally set to True to ensure the form doesn't fall behind the target object. However, FocusOnControl needs access to the target object, so the method toggles the AlwaysOnTop property, turning it off at the beginning and resetting it at the end.

Here's the code in FocusOnControl. AllControls is a cursor created by the Control Renamer engine. It contains the list of controls on the target object, along with some additional information about each of them:

```
LOCAL oControl, cControlPath, nMouseFound, nTop, nLeft
LOCAL nListIndex

This.AlwaysOnTop = .F.

IF EMPTY(AllControls.mFullPath)
  oControl = ThisForm.oChangeObject
ELSE
  cControlPath = "ThisForm.oChangeObject" + ;
    SUBSTR(AllControls.mFullPath, ;
           AT(".", AllControls.mFullPath))
  oControl = EVALUATE(cControlPath + "." + ;
    AllControls.cControlOrig)
ENDIF

IF PEMSTATUS(oControl, "SetFocus", 5) AND ;
   NOT (PEMSTATUS(oControl, "Enabled", 5) AND ;
```

```
   oControl.Enabled = .F.)
  oControl.SetFocus()
ELSE
  * Use brute force
  * First hold current mouse pos
  nMouseFound = AMOUSEOBJ(aMousePos, 1)

  IF PEMSTATUS(oControl, "Left", 5)
    * Has position properties, so click it
    * First, get position on form, not in container
    nTop = OBJTOCLIENT(oControl, 1)
    nLeft = OBJTOCLIENT(oControl, 2)
    ACTIVATE WINDOW (ThisForm.oChangeForm.Caption)
    MOUSE CLICK AT nTop + 5, nLeft + 5 PIXELS ;
      WINDOW (ThisForm.oChangeForm.Caption)
    ACTIVATE WINDOW (This.Name)
    * Reactivate this form
    MOUSE CLICK AT This.Top + 5, This.Left + 5 PIXELS

    IF nMouseFound = 4
      * Put mouse pointer back where it belongs
      DO CASE
      CASE WEXIST(aMousePos[2].Name)
        MOUSE AT aMousePos[4], aMousePos[3] PIXELS ;
          WINDOW (aMousePos[2].Name)
      CASE WEXIST(aMousePos[2].Caption)
        MOUSE AT aMousePos[4], aMousePos[3] PIXELS ;
          WINDOW (aMousePos[2].Caption)
      OTHERWISE
        * Can't do anything about it
      ENDCASE
    ENDIF
  ENDIF

ENDIF

This.AlwaysOnTop = .T.
```

Unfortunately, this code doesn't work exactly as you'd expect. There are two problems. The first is a bug in VFP. When you call the SetFocus method of the Container class at design time, focus lands on the first control inside the container, rather than on the container itself. I haven't found a way around this bug.

The other behavior probably isn't a bug, but is a problem for this tool. When you force a click on some controls that don't include a SetFocus method, it's difficult to get them to relinquish focus. Thus, as you move through the list box, focus on the form doesn't always move with you. I haven't found a workaround for this problem, either, short of only setting focus to controls with a SetFocus method. However, you can manually click on any control on the target object to re-synch.

Other form code

The only other form method containing code is Init, which instantiates the Control Renamer engine and calls its GrabControls method to build the list of controls on the target object:

```
LOCAL aProgs[1], nStackCount, cFolder

nStackCount = ASTACKINFO(aProgs)
cFolder = JUSTPATH(aProgs[nStackCount, 2])
This.oChangerEngine = NEWOBJECT("ControlRenamerEngine", ;
  FORCEPATH("ControlChanger.PRG",cFolder))

IF VARTYPE(This.oChangerEngine)<>"O"
  MESSAGEBOX("Unable to start Control Renamer Builder.")
  RETURN .F.
ELSE
  This.oChangeForm = This.oChangerEngine.oDesignerForm
  This.oChangeObject = This.oChangerEngine.oObject
  This.cObjectInfo = This.oChangerEngine.GetInfo( This.oChangeObject )

  This.oChangerEngine.GrabControls()

  This.FocusOnControl()

  This.txtCaption.Refresh()
  This.txtNewName.Refresh()
ENDIF
```

The code expects to find the Control Renamer form and the engine class in the same directory. Rather than hard-coding the directory name, it determines the form's location and looks for the engine class there.

There's very little additional code in the form and only a little of that is interesting. The InteractiveChange method for the list box refreshes other controls and calls FocusOnControl. The LostFocus method for the New Name text box checks that the newly specified name is a valid name for an object in VFP.

The Rename button's Click method ensures all the names specified are unique within their containers. If so, it calls the engine's ChangeNames method to perform the renaming:

```
IF ThisForm.oChangerEngine.CheckForDups()
  ThisForm.oChangerEngine.ChangeNames()
  ThisForm.Release()
ELSE
  MESSAGEBOX("New names include duplicates. " + ;
    "Please recheck.", 0+48,"Control Renamer")
ENDIF
```

The Control Renamer engine class

The hard work of renaming the controls in the target object is done by the Control Renamer engine class, called ControlRenamerEngine. It's based on the Custom class. ControlRenamerEngine operates in two phases. The first, performed early in the process, builds the list of controls on the target object.

The second phase, changing the code and the controls, begins when the user clicks on the Rename button on the Control Renamer form. The rename process has three steps: building a list of references to the target object's controls in the target object's code, modifying those references to use the new names, and changing the names of the controls.

ControlRenamerEngine has six custom properties (table 2).

Table 2: Engine properties–The Control Renamer engine class uses six custom properties.

Property	**Purpose**
aProcLines[1]	Holds the individual lines of a method for parsing
cOldExact	Holds the value of SET("EXACT") when the class is called
lIsForm	Indicates whether the target object is based on Form or some other base class
nCurLine	Indicates which line in aProcLines is currently being parsed
oDesignerForm	Object reference to the Form Designer or Class Designer form for the target object
oObject	Object reference to the target object

ControlRenamerEngine has quite a few custom methods. The two key methods are GrabControls, which builds the list of controls, and ChangeNames, which performs the renaming. I describe them later in this article, along with the methods they call.

Set-up and clean-up

The Init method of ControlRenamerEngine sets things up for both phases. It creates two cursors and calls the custom GetTarget method to acquire object references to the target object and the Designer form. The first cursor, AllControls, holds the list of controls on the target object. The second, ControlRefs, holds a list of references to the controls in the target object's code. Here's the code in the Init method:

```
* Create cursors
CREATE CURSOR AllControls ;
  (iID I AUTOINC, cControlOrig C(128), ;
   cControlNew C(128), mFullPath M, ;
   cInfo C(254), lReadOnly L)
INDEX on iID TAG iID
INDEX on UPPER(cControlOrig) TAG Name
SET ORDER TO

CREATE CURSOR ControlRefs ;
  (iID I AUTOINC, iRefControlFK I, iCodeControlFK I, ;
   cMethod C(128), nLine N(6), nOccurrence N(3))
* Create index so occurrences are processed
* from right to left
INDEX ON UPPER(cMethod) + TRANSFORM(nLine) + ;
      TRANSFORM(1000-nOccurrence) TAG InnerFirst

* Settings
This.cOldExact = SET("Exact")
SET EXACT OFF

* Get a reference to the containing object
This.oObject = This.GetTarget()
IF UPPER(This.oObject.Baseclass) = "FORM"
  This.lIsForm = .T.
  This.oDesignerForm = This.oObject
ELSE
  This.lIsForm = .F.
  This.oDesignerForm = This.oObject.Parent
ENDIF

RETURN
ENDPROC
```

The GetTarget method uses ASELOBJ() to get a reference to the selected object or form and traces upward through the containership hierarchy to find the containing form or class.

For forms the task is simple: Climb through Parent references until the base class of the object you're looking at is Form. However, for classes other than form classes, it isn't that easy. When you open a class in

the Class Designer, VFP creates both a formset and a form. As you move up through the hierarchy, you eventually find a form, but the object you want is the child of that form. It turns out you can distinguish a real form or form class from the form created by the Class Designer by checking for the presence of a BufferMode property. Real forms, whether SCX-based or VCX-based, have such a property, while the Class Designer's form doesn't. Here's the code for GetTarget:

```
* Grab a reference to the top-level control/form
LOCAL aSelected[1], oTarget, oLast

IF ASELOBJ(aSelected) = 0
  IF ASELOBJ(aSelected, 1) = 0
    oTarget = .null.
  ENDIF
ENDIF

IF VARTYPE(aSelected[1]) = "O"
  oTarget = aSelected[1]
  DO WHILE UPPER(oTarget.Parent.BaseClass)<>"FORMSET"
    oLast = oTarget
    oTarget = oTarget.Parent
  ENDDO

  * Form class/form or something else? If not a form
  * class need to go back down one level
  IF NOT PEMSTATUS(oTarget, "BufferMode", 5)
    * It's not a real form; it's the pseudo-form
    * represented by the Class Designer
    oTarget = oLast
  ENDIF
ELSE
  oTarget = .null.
ENDIF

RETURN oTarget
```

The Destroy method restores the original value of SET EXACT. Because the engine runs in the private data session of the Control Renamer form, there's no need to close the cursors created by the engine.

Building a list of controls

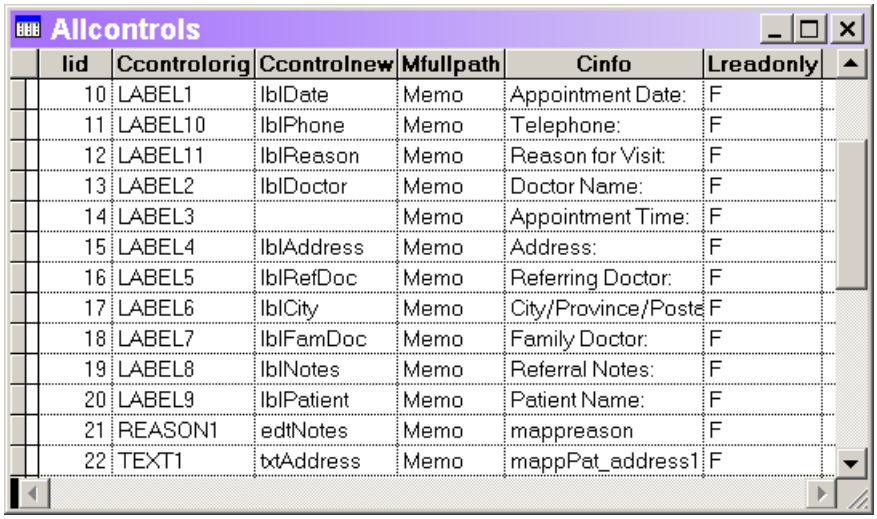
The first phase, creating a list of controls, is the simpler one. Starting from the target object, as each control is identified, a record is added to AllControls. The cursor tracks the original name of the control; the user's new name for it; a piece of identifying information, if one is available; the complete containership path to the control; and whether the control's name can, in fact, be changed.

To more easily determine the name for a control and which control's name is being specified (especially when there are focus problems on the target object), the Control Renamer form displays one identifying item for the control. The information depends on the type of control. The Caption property is used for those controls that have one. If there's no Caption property, the Value property is checked; if it exists and isn't empty, it's used. If there's no Caption and no Value, ControlSource is used, if it exists. For the remaining controls, no identifying information is provided.

To properly change references to controls in code, the complete path from the target object to the control is required. After all, a form or class can contain a dozen different Text1 text boxes at different points in the containership hierarchy.

When a control is part of a composite class, you can't change its name in forms and classes that use the composite class. For example, if you build a container class for address information, with an edit box for the street address, and text boxes for the city, state, and ZIP code, when you drop that container on a form, you can only change the name of the container. (If you could change the names of the edit box and the text boxes, code in the container class might not work.) So, the code that builds the list of controls checks for this situation and marks those controls as read-only in AllControls.

Figure 2 shows part of the contents of AllControls for a real form after new names have been specified.



lid	Ccontrolorig	Ccontrolnew	Mfullpath	Cinfo	Lreadonly
10	LABEL1	lblDate	Memo	Appointment Date:	F
11	LABEL10	lblPhone	Memo	Telephone:	F
12	LABEL11	lblReason	Memo	Reason for Visit:	F
13	LABEL2	lblDoctor	Memo	Doctor Name:	F
14	LABEL3		Memo	Appointment Time:	F
15	LABEL4	lblAddress	Memo	Address:	F
16	LABEL5	lblRefDoc	Memo	Referring Doctor:	F
17	LABEL6	lblCity	Memo	City/Province/Poste	F
18	LABEL7	lblFamDoc	Memo	Family Doctor:	F
19	LABEL8	lblNotes	Memo	Referral Notes:	F
20	LABEL9	lblPatient	Memo	Patient Name:	F
21	REASON1	edtNotes	Memo	mappreason	F
22	TEXT1	txtAddress	Memo	mappPat_address1	F

Figure 2: Auditing controls -- The AllControls cursor contains the list of controls found, including their original names and the newly specified names.

The GrabControls method starts the process of building the list. It calls DrillControls, which proceeds recursively, drilling down from the target object. After the list is built, a query fills a cursor with the list of unique control names; other methods use this list when searching for code to change. Here's the code for GrabControls:

```
* Traverse the form/container and populate
* the cursor of controls
LOCAL cInfo

* Add the form itself, then drill down
cInfo = This.GetInfo(This.oObject)
INSERT INTO AllControls ;
  (cControlOrig, mFullPath, cInfo, lReadOnly) ;
  VALUES (This.oObject.Name, "", m.cInfo, .F.)

This.DrillControls(This.oObject, This.oObject.Name)

* Now get a list of unique names
SELECT DISTINCT cControlOrig ;
  FROM AllControls ;
  INTO CURSOR ControlNames

RETURN
```

The custom DrillControls method does most of the work of building the list. It uses AMEMBERS() to get a list of controls contained by the control it receives as a parameter, then processes each control in that list:

```
PROCEDURE DrillControls(oContainer, cHierarchy)
* Drill into a container and add all the controls
```

```
* inside to the cursor

LOCAL nControls, aControls[1], nControl, oObject
LOCAL nPEMs, aPEMs[1], nNameRow, lReadOnly

nControls = AMEMBERS(aControls, oContainer, 2)

FOR nControl = 1 TO nControls
  * Figure out what info is available about this control
  oObject = EVALUATE("oContainer." + aControls[nControl])
  cInfo = This.GetInfo(oObject)

  * Find out whether name can be changed
  nPEMs = AMEMBERS(aPEMs, oObject, 3, "#+")
  nNameRow = ASCAN(aPEMs,"NAME",-1,-1,1,15)
  IF nNameRow <> 0
    lReadOnly = "R"$aPEMs[nNameRow,5]
  ELSE
    lReadOnly = .T.
  ENDIF

  INSERT INTO AllControls ;
    (cControlOrig, mFullPath, cInfo, lReadOnly) ;
    VALUES (aControls[nControl], cHierarchy, ;
          m.cInfo, m.lReadOnly)

  * Drill down
  IF PEMSTATUS(oObject, "Objects", 5)
    * Drill down
    This.DrillControls(oObject, ;
      cHierarchy + "." + aControls[nControl])
  ENDIF
ENDFOR

RETURN nControls
```

Both GrabControls and DrillControls call the custom GetInfo method to return the identifying information for the current object. That method is simply a case statement and it checks the various options for additional information.

Checking the new names

The CheckForDups method, called from the Control Renamer form's Rename button, ensures the new names specified by the user result in a unique path to each control. It uses the information in AllControls. Note that AllControls.cControlNew is bound to the New Name text box in the Control Renamer form.

```
* Ensure that new names don't include duplicates

LOCAL lReturn
```

```
SELECT UPPER(IIF(EMPTY(cControlNew), ;
                 cControlOrig, cControlNew)), ;
     UPPER(mFullPath), CNT(*) ;
  FROM AllControls ;
  GROUP BY 1,2 ;
  HAVING CNT(*) > 1 ;
  INTO CURSOR Dups

lReturn = _Tally=0

USE IN Dups
RETURN lReturn
```

Finding references to controls

The BuildCodeRefs method and the methods it calls read all the code in the target object and its contained objects, and populates the ControlRefs cursor with one record for each reference to a control on the target object. Like GrabControls, this is a recursive process.

BuildCodeRefs is quite simple. It starts things off by calling the DrillCode method, passing the target object as a parameter:

```
This.DrillCode(This.oObject)
```

The real work is done in the DrillCode method and the methods it calls. DrillCode uses AMEMBERS() to find all events and methods. It calls the AuditMethod method to check each event or method for references to controls.

When that's done, DrillCode calls itself recursively for any contained objects. This is where I encountered the second VFP bug. Although Grid has an Objects collection, you can't use it to go through the list of columns, so the code handles grids separately from other container objects:

```
PROCEDURE DrillCode(oControl)

LOCAL aAllMem[1], nMembCount, nObject, nMember

* Make sure list of controls to search for exists
IF NOT USED("ControlNames")
  RETURN .F.
ENDIF

nMembCount = AMEMBERS(aAllMem, oControl, 1)
FOR nMember = 1 TO nMembCount
  IF INLIST(UPPER(aAllMem[nMember, 2]), ;
            "EVENT", "METHOD" )
    This.AuditMethod(oControl, aAllMem[nMember,1])
```

```
  ENDIF
ENDFOR

* Work around bug with Grid.Objects
DO CASE
CASE UPPER(oControl.BaseClass)="GRID"
  IF oControl.ColumnCount > 0
    FOR nObject = 1 TO oControl.ColumnCount
      This.DrillCode(oControl.Columns[ nObject ])
    ENDFOR
  ENDIF

OTHERWISE
  IF PEMSTATUS(oControl, "Objects", 5)
    FOR nObject =1 TO oControl.Objects.Count
      This.DrillCode(oControl.Objects[nObject])
    ENDFOR
  ENDIF
ENDCASE

RETURN
```

AuditMethod is the heart of the process of finding references. It reads the code for a method, then loops through the list of control names, checking each in turn. If a control name is found in the current method, the line containing it is parsed to put together the complete path to that control. If necessary, AuditMethod traces backward through the code to find a containing WITH statement. In addition, the keywords Parent, This, and ThisForm are converted to the appropriate references.

One of the biggest challenges in writing this code was finding exact control names. Because you might have Text2 and Text20 on the same form, and control names don't generally appear where exact string matching can be used, I needed another approach. I chose to use the periods on either side of a control's name as delimiters. That meant I had to deal with the possibility that the name of a control might be the last thing on a line of code. For example, consider this line:

```
oControl = This.PageFrame1.Page2.Text3
```

To handle such cases, AuditMethod breaks the code for a method into lines and adds a period at the end of each line before searching. That makes it possible to search for a control name surrounded by periods and know that it will find Text3. You can find the code for AuditMethod on this issue's Professional Resource CD.

The comments in the code point out several situations the tool can't handle. Be aware of the need to follow up with manual changes if any of them apply.

AuditMethod calls on four more custom methods: GetCodeLineByPos, FindWith, BuildControlPath, and LookUpControl. GetCodeLineByPos accepts a block of code and a numeric position, and returns the line of code that contains that position. It uses ALINES() to break the block of code into individual lines and then figures out how many line breaks (CHR(13)) occur before the specified position.

FindWith accepts a line number in the current method and searches backward (toward the beginning of the method) to find a WITH statement. Because WITH can be nested, the method is recursive:

```
PROCEDURE FindWith(nStartLine)
* Find the first occurrence of WITH preceding
* the specified line and extract the referenced object

LOCAL nLine, cWithLine, nNamePos, cObject, cContainer

nLine = nStartLine - 1
cWithLine = ALLTRIM(This.aProcLines[nLine])

DO WHILE nLine > 0 AND UPPER(cWithLine) <> "WITH"
  nLine = nLine - 1
  cWithLine = ALLTRIM(This.aProcLines[nLine])
ENDDO

IF nLine > 0
  nNamePos = AT(" ", ALLTRIM(cWithLine))
  cObject = ALLTRIM(SUBSTR(cWithLine, nNamePos + 1))
  * Remove trailing period
  IF RIGHT(cObject,1)="."
    cObject = LEFT(cObject, LEN(cObject)-1)
  ENDIF

ELSE
  cObject = ""
ENDIF

IF LEFT(cObject, 1) = "."
  * Nested WITH, keep going back
  cContainer = This.FindWith(nLine)
  cObject = cContainer + cObject
ENDIF

RETURN cObject
```

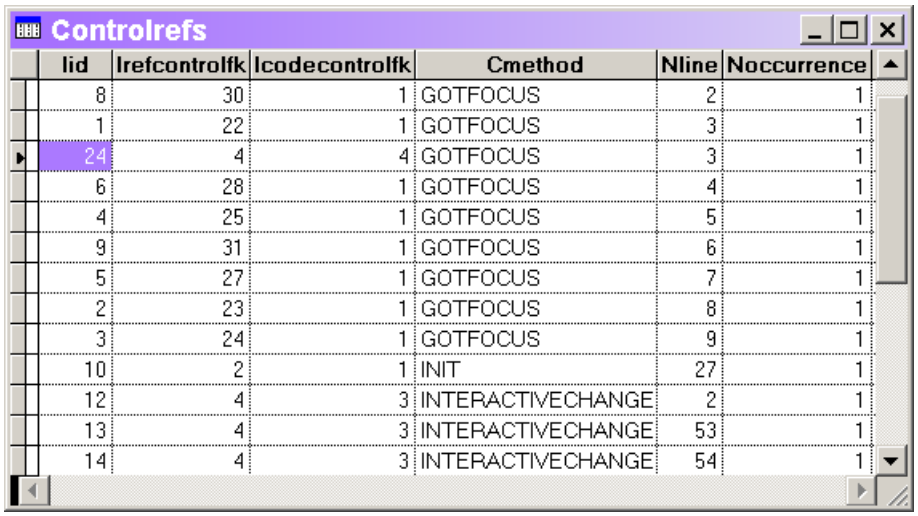
This method points out another situation the Control Renamer can't handle: the possibility of a WITH that crosses method boundaries; that

is, a reference in one method that relies on WITH having been issued in the calling method. (Of course, using such a reference is a bad idea.)

BuildControlPath accepts an object reference to a control and builds a complete path to the control by moving upward through the Parent reference.

LookUpControl accepts either an object reference to a control or its name, and looks it up in the AllControls cursor. It returns the primary key for the control in the cursor; if the control can't be found, it returns -1.

Figure 3 shows part of the contents of ControlRefs for the same form whose controls are shown in figure 2.


Controlrefs

lid	lrefcontrolfk	lcodecontrolfk	Cmethod	Nline	Noccurrence
8	30	1	GOTFOCUS	2	1
1	22	1	GOTFOCUS	3	1
24	4	4	GOTFOCUS	3	1
6	28	1	GOTFOCUS	4	1
4	25	1	GOTFOCUS	5	1
9	31	1	GOTFOCUS	6	1
5	27	1	GOTFOCUS	7	1
2	23	1	GOTFOCUS	8	1
3	24	1	GOTFOCUS	9	1
10	2	1	INIT	27	1
12	4	3	INTERACTIVECHANGE	2	1
13	4	3	INTERACTIVECHANGE	53	1
14	4	3	INTERACTIVECHANGE	54	1

Figure 3: Tracking references in code—The ControlRefs cursor has one record for each mention of any control in the target object's code. The foreign keys point to the AllControls cursor; the record indicates what line of what method contains the reference.

Changing code

The ChangeNames method is the main routine of the rename phase. It calls BuildCodeRefs to populate the ControlRefs cursor. Then, it makes the changes. The InnerFirst index tag is used so that changes proceed from right to left within any particular reference. For example, given this line:

```
ThisForm.PageFrame1.Page1.Text1
```

the tag ensures Text1 is changed first, then Page1, then PageFrame1. Changing the names in any other order makes it extremely difficult to find all the changes.

Like AuditMethod, this code adds a period to the end of a line before performing the replacement. The extraneous period is removed after the replacement before adding the changed line to the overall method code.

After all the code has been changed, the method changes the name of the controls. You can find code demonstrating this on this issue's Professional Resource CD.

Putting the Control Renamer to work

There are two ways to use the Control Renamer. The easier approach is to open the target object in the Form Designer or Class Designer and then issue DO FORM RenameControls.

However, you can also use the tool as a builder by registering it in Wizards\Builder.DBF: the table that tracks available builders for VFP's builder system. After you do this, use it by right-clicking in any form or class and choosing Builder. In some cases, a dialog appears, offering you a choice of builders. (In addition, because it's registered as an "ALL" builder, the dialog appears at some unlikely times, such as when you add a PEM to Favorites in VFP 9. In that situation, choose MemberData Editor from the dialog.)

This issue's Professional Resource CD includes ccBuilderMain.PRG, a program that registers the builder if it isn't already registered, and then runs it. Run this program once from the Command window to register the builder, and from then on it's available. (I described this technique in my article "Build Your Own Builders" in the August 2002 issue at http://My.Advisor.com/doc/09950.)

After you have the Control Renamer running, you can select each control and specify a new name for it. (You don't have to rename every control, though; the tool renames only those for which you specify a new name.) When you have the names you want, click on the Rename button and the tool does the rest.

The perfect solution

In a perfect world, every VFP developer would give every control a meaningful name before writing code. But in the real world, even good

developers sometimes forget. With the Control Renamer, fixing the problem is simple.