August, 2002

Build Your Own Builders

Builders can make development of forms and classes easier than ever

By Tamar E. Granor

One of the most underused tools in Visual FoxPro is the Builder system. That's probably because the builders that come with VFP make it look like all you can do with builders is format controls the way you want them. But that's a massive understatement.

A builder is any program that manipulates a visual object at design-time. While builders can and often do have polished user interfaces, it's also possible to have a builder with no user interaction at all.

The secret to builders is that VFP's forms and classes can be modified programmatically at design-time. Pretty much anything you can change in the Property Sheet, the Method Editor, or by dragging things around, you can also do programmatically.

Getting a handle on forms and controls

The first key ingredient to creating builders is the ability to get references to the objects that are being designed. The ASelObj() function fills an array, with one item for each selected object. (Get it- "Array of SELected OBJects.") Each item in the array is an object reference, so once you issue ASelObj(), you can touch the PEMs of each selected object and, of course, once you have access to the controls, you can also get to the form that contains them (or any other intermediate containers).

To see how easy it is, try this. Create a new form by issuing CREATE FORM. Drop a few controls onto it. Select all the controls (by issuing Edit-Select All or typing CTRL+A). From the Command Window, issue:

```
? ASelObj(aAllControls)
```

The number of controls you selected is echoed to the main VFP window. Now you can see what they are, using code like this:

```
FOR EACH oControl IN aAllControls
   ?oControl.Name
ENDFOR
```

But you're not limited to asking about the controls; you can change them, too. This code aligns the controls vertically:

```
FOR EACH oControl IN aAllControls
   oControl.Left = 10
ENDFOR
```

Reading and setting properties

As the examples above show, reading and setting properties programmatically is no harder at design-time than at runtime.

However, there's one complication. Sometimes, a property contains an expression meant to be evaluated at runtime rather than at design-time. To enter such an expression in the property sheet, you precede it with "=". For example, the Caption property for a label might include the date, using an expression like:

```
="Today is " + TRANSFORM(DATE())
```

Checking such an expression directly evaluates it. So, if this label is the first control in aAllControls, code like the following:

```
? aAllControls[1].Caption
```

prints out the resulting value, not the expression:

```
Today is 01/02/02
```

Fortunately, there is a way to retrieve the expression. The ReadExpression method returns as a string the expression in a property. Continuing the example, this call:

```
? aAllControls[1].ReadExpression( "Caption" )
```

prints out the original expression:

```
="Today is " + TRANSFORM(DATE())
```

If a property contains a value rather than an expression, ReadExpression returns the empty string, so it's easy to distinguish the two cases.

In a builder, we might also want to set a property to an expression. Setting the property to the expression directly evaluates the expression. For example, this line:

```
aAllControls[1].Caption = "Today is " + TRANSFORM(DATE())
```

sets the Caption to the evaluated result:

```
"Today is 01/02/02"
```

Setting it to the expression as a character string gives the property a character value. For example, this line:

```
aAllControls[1].Caption = '"Today is " + TRANSFORM(DATE())'
```

results in a caption of:

```
"Today is " + TRANSFORM(DATE())
```

Again, you need a special method. The WriteExpression method stores an expression to a property without evaluation. So, the right way to set this caption is:

```
aAllControls[1].WriteExpression("Caption",;
   '"Today is " + TRANSFORM(DATE())')
```

Reading and setting methods

Handling code at design-time is actually simpler than handling properties. The ReadMethod and WriteMethod methods let you extract and store code, respectively.

For example, to retrieve and display the code in the Click method of the second selected control in our example, use this code:

```
? aAllControls[2].ReadMethod("Click")
```

Storing code is a little harder because you have to assemble it first. This very simple example puts code in that Click method to display a WAIT WINDOW.

```
LOCAL cCode
cCode = "WAIT WINDOW 'Clicked me, did you?'"
aAllControls[2].WriteMethod( "Click", cCode )
```

Of course, usually, the code you want to store has more than one line. Separate the lines with CHR(13) (the return character).

In VFP 7, WriteMethod has one additional feature. You can use it to add new methods to an object. Pass .T. for the optional third parameter and, if the method you specify doesn't already exist, it's added and populated with the specified code.

Finding the parent or form

For some operations, you need to find the container of a control or the form on which the control sits. To get to the parent, use the Parent property of the object. This example finds the parent of the first selected control from our example:

```
IF NOT ISNULL(aAllControls[1].Parent)
   oParent = aAllControls[1].Parent
ENDIF
```

A similar strategy works for finding the form, but in this case, you need to climb the containership hierarchy until you find a form or run out of levels (as you might in the Class Designer). This code finds the form containing the first selected control in the example.

```
oObject = aAllControls[1]
DO WHILE NOT ISNULL(oObject.Parent) AND ;
         UPPER(oObject.Parent.BaseClass)<> "FORM”
   oObject = oObject.Parent
ENDDO
oForm = oObject.Parent
```

There's actually another way to get your hands on the containers of the selected objects. Pass the number 1 for the optional second parameter of ASelObj() and, instead of object references to the controls, you get object references to their containers. You can also pass the number 2 for that parameter to get a reference to the data environment of the containing form.

Adding controls

Some builders (including the one described below) need the ability to add controls to a form or class. The AddObject method of all container objects makes this simple. Pass the name and class (and, optionally, class library) of the object to be added.

For example, to add a label called lblNew, based on the Label base class, to the form referenced by oForm:

```
oForm.AddObject("lblNew","Label")
```

Building a builder

With all these tools in hand, we're ready to build a useful builder.

Have you ever worked on a form or class and, after doing quite a bit of work, discovered that you were dropping controls from the wrong class

library, maybe even from the FoxPro base classes? I sure have. So I wrote a builder that lets me replace any control with another control descended from the same base class without losing any properties I've set or method code I've written. While it's possible to fix this problem by opening the form or class library as a table and modifying the data, I don't have to leave the Form or Class Designer to fix the problem. Second, modifying an .SCX or .VCX directly always opens up the possibility of fouling it up; if I can solve the problem another way, I'd rather.

This builder (available on this month's Professional Resource CD and from Advisor.COM) is based on a form class because it provides an interactive means of choosing the new class. (Be aware, that this builder requires VFP 7 because it uses some of the new features.) The Init method of the form checks to make sure there's exactly one control selected and stores a reference to that control in the form's custom oControl property.

The key method of the builder is ReplaceControl. It adds a new control to the original control's container, then goes through all the PEM's of the original control, copying any changed values to the new control. Finally, it removes the original control and selects the replacement, in order to leave things as it found them. (Unfortunately, there's a bug in VFP 7 Service Pack 1 that leaves the property sheet showing "Multiple Selection" rather than the new object selected. Microsoft is aware of the problem.)

There are a few limits on what gets copied. First, if the property or method is protected in the new control, its contents in the original control are not copied. That's because you can't write to protected properties and methods.

In addition, custom properties and methods of the original control are not added to the new control. Since you can't add properties to a control once you put it inside a container (like a form or another class), any such properties and methods must have been added to the class of the original control (or one of its ancestor classes). If you're changing the underlying class, then presumably you don't want the special features of the original class.

In order to ensure that you don't lose any code, the builder saves the contents of any custom methods of the original class that don't exist in the replacement class and of any methods of the original class that are protected in the replacement class. This code is saved in a text file and placed in a MODIFY FILE editing window that's left open by the builder.

Here's the code for ReplaceControl:

```
* Replace the specified control with
* a control of the chosen class.

LOCAL oContainer, oOriginalControl, aOldProps[1]
LOCAL cSavedCode, oForm

* Make sure the new class name and class library
* are in the right properties
This.Getnewclass()

* First, get a reference to the containing object
oContainer = This.oControl.Parent

* Save a reference to the original control
oOriginalControl = This.oControl

* Add the new control
cTempName = SYS(2015)
oContainer.NewObject( cTempName, ;
   This.cNewClass, This.cNewClassLib )
oControl = EVALUATE( "oContainer." + cTempName)

* Copy properties
* Get properties for the original object
* Note work-around in the next line, using "#+"
* for flags, when only "#" is needed. "#" alone
* is not accepted by AMEMBERS().
nOldMembers = AMEMBERS(aOriginalProps, ;
                       oOriginalControl, 3,"#+")
cSavedCode = ""
FOR nMember = 1 TO nOldMembers
   DO CASE
   CASE "R" $ UPPER(aOriginalProps[ nMember, 5])
      * Read-only, so skip it
   CASE "NAME"=UPPER(aOriginalProps[ nMember, 1])
      * Name property, we'll do it later
   CASE "PROPERTY" $ ;
        UPPER(aOriginalProps[ nMember, 2])
      * See whether the property has changed
      * and the new control has this property.
      * If so, copy it.
      * Protected properties aren't included
      * because you can't write to them.
      IF "C" $ UPPER(aOriginalProps[ nMember, 5]) ;
         AND PEMSTATUS( oControl, ;
                aOriginalProps[ nMember, 1], 5) ;
         AND NOT PEMSTATUS( oControl, ;
                    aOriginalProps[ nMember, 1], 2)
         cPropName = aOriginalProps[ nMember, 1]
         IF EMPTY(oOriginalControl.ReadExpression( ;
                  cPropName))
            oControl.&cPropName = ;
               oOriginalControl.&cPropName
```

```
         ELSE
            * The property contains an expression
            oControl.WriteExpression(cPropName, ;
               oOriginalControl.ReadExpression(;
               cPropName))
         ENDIF
      ENDIF
   CASE INLIST(UPPER(aOriginalProps[ nMember, 2]), ;
               "METHOD", "EVENT")
      * See whether the method has changed
      * and whether the new control has this method
      * If so, copy the contents
      IF "C" $ UPPER(aOriginalProps[ nMember, 5])
         cMethod = aOriginalProps[ nMember, 1]
         cMethodCode = oOriginalControl.ReadMethod( ;
            cMethod)
         IF NOT EMPTY(cMethodCode)
            * Some methods show as changed even though
            * they have no code at this level
            IF PEMSTATUS( oControl, ;
                  aOriginalProps[ nMember, 1], 5) ;
               AND NOT PEMSTATUS( oControl, ;
                           aOriginalProps[ nMember, 1], 2)
               oControl.WriteMethod( cMethod, ;
                                     cMethodCode )
            ELSE
               cSavedCode = cSavedCode + ;
                  "* Code from " + cMethod + CHR(13)
               cSavedCode = cSavedCode + ;
                  cMethodCode + CHR(13)
            ENDIF
         ENDIF
      ENDIF
   OTHERWISE
      * We should never get here
   ENDCASE
ENDFOR

* Put the saved code, if any, in an editing window
IF NOT EMPTY(cSavedCode)
   cCodeFile = ADDBS(SYS(2023)) + "SavedCodeFrom" + ;
               oOriginalControl.Name + ".TXT"
   STRTOFILE( cSavedCode, cCodeFile, 1)
   MODIFY FILE (cCodeFile) NOWAIT
ENDIF

* Remove original object
cOldName = oOriginalControl.Name
oContainer.RemoveObject( oOriginalControl.Name )

* Rename new object
oControl.Name = cOldName

* Select newly added object
IF PEMSTATUS(oControl,"SetFocus",5)
```

```
   oControl.SetFocus()
ELSE
   * Use brute force method
   * Find containing form
   oForm = oContainer
   DO WHILE NOT ISNULL(oForm) ;
      AND UPPER(oForm.BaseClass) <> "FORM"
      oForm = oContainer.Parent
   ENDDO
   IF PEMSTATUS(oControl,"Left",5)
      * Click on the new control
      MOUSE CLICK AT oControl.Top+1, oControl.Left+1 ;
            PIXELS WINDOW (oForm.Name)
   ELSE
      * Last chance. Set focus to form
      MOUSE CLICK AT 1,1 WINDOW (oForm.Name)
   ENDIF
ENDIF

RETURN
```

The version of the builder (Figure 1) on this month's Professional Resource CD lets you choose the new class from among those registered in the Component Gallery, using a custom control. But the builder is designed to make it easy to change the manner of choosing a new control. References to the custom control are isolated in the SetupChoice, EnableControls and GetNewClass methods.

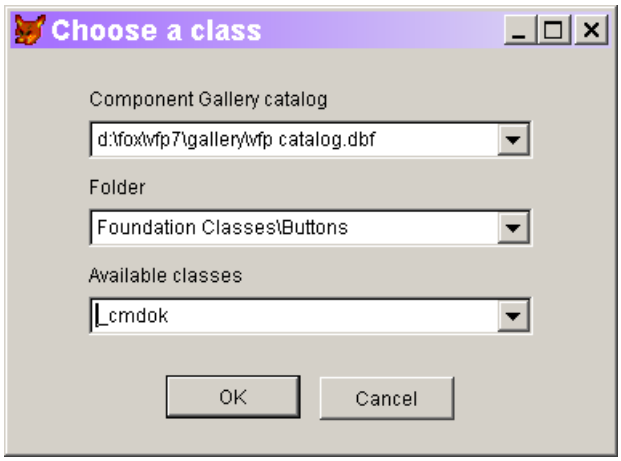

Figure 1. The Base Class Builder lets you substitute one control for another on the fly.

SetupChoice is called from the Init method to perform any tasks relating to class selection needed at start-up. EnableControls is used to enable and disable controls on the form-it's called from Init and also from the custom control. GetNewClass is called from ReplaceControl to

retrieve the chosen class name and library and put them into properties.

To substitute a different way of choosing a class, simply replace the custom control with something else and modify those three methods appropriately.

Putting a builder to work

You can use a builder by simply running it from the Command Window when you need it, but although it requires a little effort up front, the easiest way to use a builder repeatedly is to register it with the Builder system that comes with VFP. This is the application (Builder.APP) used when you choose a builder from within the Form or Class Designer (by right-clicking on a control and choosing Builder, or by setting the Builder Lock in the Form Controls Toolbar).

The Builder system stores a list of builders in Builder.DBF in the Wizards directory. To register a new builder, you simply add a record to that table. There are four key fields you need to specify:

Name–the name to use for the builder.

Descript–a longer description of the builder.

Type–what kind of objects the builder applies to.

Program–the program to run when the builder is called.

Name, Descript and Program are pretty straightforward, but Type calls for some explanation. Builders fall into three broad categories: those that apply to only a single base class; those that can operate on a group of selected objects; and those that can be used for objects of any base class.

For builders that work with only a single base class, specify the name of the class for Type. For builders that can modify multiple selected objects, specify "MULTISELECT" for Type. Specify "ALL" for builders that work with objects from any base class (such as the builder above).

If you want, you can simply open the Builder table and add a record manually to register a builder. However, there's another approach (that I learned from Doug Hennig) that both documents the registration and makes it easier to register a builder on many machines. The trick is to write a main program that registers the

builder, if necessary, then executes it. Here's such a program (BaseBuilderMain2.PRG on this month's PRD) for the builder above:

```
* Main program for BaseBuilder2

LPARAMETERS uP1, uP2, uP3, uP4, uP5, uP6, ;
            uP7, uP8, uP9, uP10, uP11, uP12
* Accept parameters passed by the builder system

#DEFINE ccMAIN    "BASEBUILDERMAIN2"

LOCAL nOldSelect

* Self-register if called directly
IF PROGRAM(0) == ccMAIN
    nOldSelect = SELECT()
    SELECT 0
    USE HOME() + "Wizards\Builder" AGAIN
    LOCATE FOR Name = "Base Class Builder"
    IF NOT FOUND()
        m.Name = "Base Class Builder"
        m.Descript = ;
           "Choose a class for base class controls"
        m.Type = "ALL"
        m.Program = SYS(16)
        INSERT INTO Builder FROM MEMVAR
    ENDIF

    USE IN Builder
    SELECT (nOldSelect)
ENDIF

* Run the actual builder
DO FORM ADDBS(JUSTPATH(SYS(16))) + "BaseBuilder2"

RETURN
```

Builders called by the Builder system must accept up to 12 parameters. As far as I can tell, once you accept them, you can simply ignore them. A look inside the Builder application indicates that at least part of what was intended was never implemented.

To register the builder, call this program directly from the Command Window. If the builder isn't already registered, it adds a record to the Builder table. Then, it calls the actual builder form. (If you prefer, you can hide some detail by building an APP with the program set as the main program. Then, just run the APP to register the builder.)

Once the builder is registered, it's available all the same ways as the builders that come with VFP.

To test the base class builder, create a form and drop a control on it. Change some properties of the control and add some code to a method. Right-click on the control and choose Builder. In the form that appears, choose another control of the same base class (those are the only ones that appear). If you haven't added your own classes to the Component Gallery, try looking in the catalog "VFP Catalog" for replacement classes. (The "VFP Catalog" doesn't contain items descended from every base class, so if you see "(None)" and "(No classes)" in the dropdowns, try testing with a different control.)

Build your own

What builders should you create for yourself? Watch yourself work in the Form and Class Designers. Each time you find a task that you seem to do over and over pretty much the same way, think about whether it can be automated. Spending a little time writing a builder can save you hours or even days in the long run.

Sidebar: Making it even easier

There are a couple of tools available that simplify the process of building your own builders. BuilderB is a "builder builder." It walks you through the process of creating a builder for a particular control class. BuilderB includes a builder form class and builder control classes to use in builder forms. BuilderB is included in the downloads available at http://www.advisor.com/Articles.nsf/aid/GRANT96. There's a text file that walks you through the process.

But wait, there's more. BuilderD is a "data-driven builder builder." It creates builder forms on the fly based on data stored in a table. BuilderD comes with VFP 6 (and is used by the FoxPro Foundation Classes). Look for BuilderD.VCX in the Wizards directory. The best documentation I know of for using BuilderD is Doug Hennig's white paper on developer tools, available at http://www.stonefield.com/techpap.html.