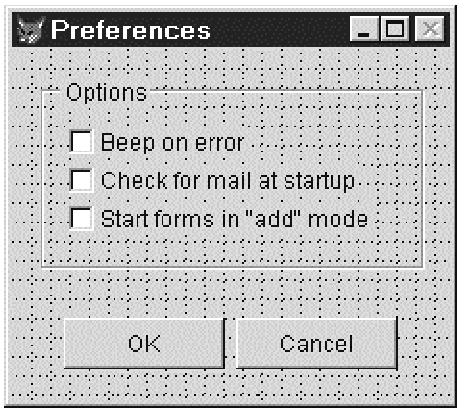
# uilding Your Own Builders with BuilderB

*Doug Hennig and Yuanitta Morhart*

**Builders make it easy to set properties for objects at design time, which is especially handy for containers which you normally have to drill down into. Ken Levy's BuilderB tool makes creating your own builders almost a trivial job.**

I like to use a 3-D box to group controls with a common purpose, and provide a label that sits on the box indicating what the controls are for. For example, in Figure 1, the box labeled "Options" groups the check boxes, making it obvious that they belong together and provide options the user can choose from.

*Figure 1. SFLabelledBox control.*



Because a labeled box is something I envision using often enough, it was worth it to create a labeled box control. This was very easy to do:

- I created a new class in SFCCTRLS.VCX (our class library of custom controls) called SFLabelledBox based on SFContainer (our base class for all containers, which is defined in SFCTRLS.VCX).
- I added an SFShape object to the container, named it shpBox, and sized and positioned it as follows: Top = 6 (to leave room for the label), Left = 0, Width = width of the container, and Height = height of the container - 6.
- I added an SFLabel object to the container, named it lblLabel, and set the following properties: BackStyle = 1 (Opaque; although this is the default for the VFP Label base class, SFLabel changes this to Transparent since that's what we normally want, so we need to change it back for this particular instance), Caption = " Box Label " (with a space at the start and end of the text so there's some room around the label when it sits on the box), Top = 0, and Left = 10.

Now, whenever I need a labeled box, I simply drop one from SFCCTRLS onto a form or class. Simple, right? Well, there's one slight annoyance: containership. The same concept that makes it easy to move this control around makes it more difficult to change properties of the individual components. For example, when you resize the SFLabelledBox object, you're really just resizing the container. You then have to right-click on the container, choose Edit in the shortcut menu, click on the shpBox object, and resize it, ensuring that you use the proper dimensions to match the container. Similarly, you have to drill down into the container to get at the caption of the label. It almost seems easier to drop a shape and label on a form and manipulate them individually than to go through these machinations with SFLabelledBox. That is, until you remember that VFP allow you to create custom builders that can make changing properties of objects at design time a snap.

## Creating a Builder

Creating a builder isn't difficult, but can be a fair amount of work, and may not seem like it's worth the effort for classes you may not use very often. Fortunately, Ken Levy has created and released into the public domain a builder builder (I'll try to resist making too many of the obvious jokes that come out of this situation <g>) called BuilderB. Although BuilderB is included in this month's Subscriber Download files, you can get the latest copy of BuilderB from Ken's Web site: www.classx.com.

BuilderB consists of several classes that allow you to quickly create a standardized builder for just about any object. One look at all those classes may make you think using BuilderB is a daunting task, but in fact you only use a handful of classes, and you generally only set one or two properties for each of those classes and probably won't enter even a single line of code.

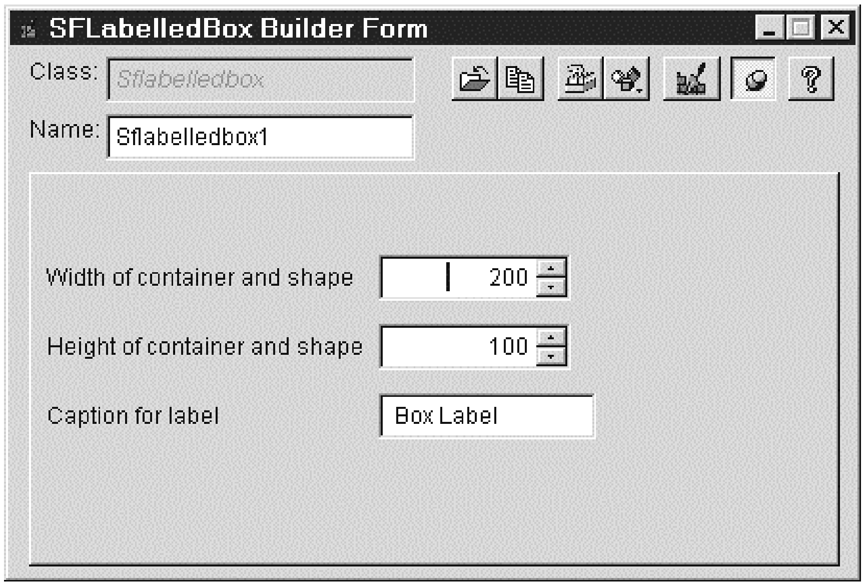
Yuanitta Morhart, a software developer with Stonefield Systems Group, was particularly enthused with BuilderB after she attended Ken's session on it at Great Lakes Great Database Workshop in March 1997. I'm going to turn things over to her to describe how she used BuilderB to create a builder for the SFLabelledBox class.

## How I Built a Builder (Yuanitta)

I'm going to describe the step-by-step process that I followed to create a builder for the SFLabelledBox class. BuilderB includes a read me file called BUILDERB.TXT in which Ken describes how to use BuilderB. In general, I followed his instructions, but made a few changes along the way.

My goal was to create a builder which would allow me to do the following: drop an SFLabelledBox object on a form, right-click on the object, choose Builder from the shortcut menu, fill in the width and height, specify a caption for the label, and choose OK. The builder should adjust the width and height for both the container and the shape, accounting for the fact that they have slightly different heights. Figure 2 shows the finished product so you get an idea of how the builder will look.

*Figure 2. BuilderB builder for the SFLabelledBox control.*



Class: Sflabelledbox
Name: Sflabelledbox1
Width of container and shape: 200
Height of container and shape: 100
Caption for label: Box Label

I started by entering "SFBldrs,SFLabelledBoxBuilder" into the Builder property of the SFLabelledBox class. Builder is a custom property we added to every class defined in SFCTRLS.VCX. This property, which is blank by default, tells the VFP BUILDER.APP what class to use as the specific builder for this class (if this property doesn't exist or is blank, VFP uses the default builder for the BaseClass of the object). This is a much simpler mechanism than VFP 3, which required you to register the builder in BUILDER.DBF and didn't allow you to create specialized builders for particular classes (although another Ken Levy utility, BuilderX, provided a way around this problem). The text before the comma is the name of the VCX containing the class, and the text following the comma is the name of the class in that VCX. Thus, we're going to have an SFBLDRS.VCX class library containing a class called SFLabelledBoxBuilder that will be the builder for SFLabelledBox objects.

Next, I created a new class in SFBLDRS.VCX (which didn't exist, so VFP automatically created it) called SFLabelledBoxBuilder, based on the BuilderFormClass contained in BUILDERB.VCX. BuilderFormClass has all the properties and methods needed in a builder, so you don't have to worry about the underlying mechanism of how a builder gets connected to the selected object or controls its properties. Although I could've put SFLabelledBoxBuilder in SFCCTRLS.VCX, it made more sense to put all the builders I'll create into their own class library.

The next step is to add some controls to SFLabelledBoxBuilder. I wanted a text box for the caption of the label and two spinners, one for the height of the SFLabelledBox object and one for the width. BUILDERB.VCX has most of the controls you'll need to use on a builder form, except for a spinner, so I had to create one. I didn't want to add a new class to BUILDERB.VCX (downloading a new version of BUILDERB.VCX from Ken's Web site would overwrite any changes I made), so I created a new class in SFBLDRS.VCX called BuilderSpinner, based on the VFP Spinner base class. I copied all of the methods and properties from the BuilderTextBox class to BuilderSpinner (*Doug: times like this make me wish VFP supported multiple inheritance*), then changed the Init method to assign 0 (rather than an empty string) to Value and the InteractiveChange method to not ensure Value is of type Character.

After creating BuilderSpinner, I realized I had an extra complication. Each BuilderB class has a custom cProperty property. cProperty is used to tie a control in a builder to a particular property in the object being maintained with the builder. You set cProperty to the name of the property you want to adjust; for example, use "Width" so changes in the control affect the Width property. However, I wanted to use one spinner to affect a property of two different objects: the SFLabelledBox container and the shpBox shape it contains. In order to do this, I created a subclass of BuilderSpinner called BuilderSpinner2Property and added a second custom property called cProperty2. I put the following code into the RefreshPropertyValue method (which is called whenever the value of the control is changed, so changes cause the property it manages to change as well):

```
local lcCurrProperty
dodefault()
lcCurrProperty = This.cProperty
This.cProperty = This.cProperty2
dodefault()
This.cProperty = lcCurrProperty
```

Because the normal RefreshPropertyValue method updates the property whose name is contained in cProperty with the value of the control, we can use DODEFAULT() to get the normal behavior, then put the contents of cProperty2 into cProperty and use DODEFAULT() again to make it update the second property. Before exiting this method, the code restores the former cProperty setting.

Once the BuilderSpinner and BuilderSpinner2Property classes were setup, I carried on with working on my builder (SFLabelledBoxBuilder). I added two instances of the BuilderSpinner2Property class (one for the width and the other for the height), one instance of BuilderTextBox (defined in BUILDERB.VCX) for the label's caption, and three instances of BuilderLabel (also in BUILDERB.VCX). I set the captions for the BuilderLabels as shown in Figure 2 and set cProperty for the BuilderTextBox to "lblLabel.Caption" so it's tied to the Caption property of the lblLabel object in the SFLabelledBox object. For the first BuilderSpinner2Property object, I set cProperty to "Width" and cProperty2 to "shpBox.Width"; this means this control will adjust the widths of the SFLabelledBox container and the shpBox object within the container.

The BuilderSpinner2Property object controlling the height is a little more complicated: while it directly affects the height of the container, the height of the box is 6 pixels less than that of the container, so we have to override the RefreshPropertyValue method for this object with the following code:

```
local lcCurrProperty, ;
  lnCurrValue
dodefault()
lcCurrProperty = This.cProperty
lnCurrValue    = This.Value
This.cProperty = This.cProperty2
This.Value     = max(This.Value - 6, 1)
```

```
dodefault()
This.cProperty = lcCurrProperty
This.Value     = lnCurrValue
```

That’s it! While this may seem like a fair amount of work, most of it was not creating the builder itself but creating some new classes for BuilderB with additional functionality. Most builders you’ll create with BuilderB will be much simpler, and probably won’t require any code at all.

## Using the Builder (Doug)

Using the builder Yuanitta created is easy. Drop an SFLabelledBox object on a form, right-click on the object, and choose Builder from the shortcut menu. As you enter the caption for the label in the text box or use the spinners to change the size of the object, notice the caption or the size changes instantly. That’s because the custom lUpdateOnChange property of the BuilderB controls is set to .T., meaning any changes are immediately reflected back to the property they control. If you want to turn this feature off, simply set the property to .F.; the property will then be updated only when the control loses focus.

Builders created with BuilderB provide additional functions:

- Changing the name of the object
- Saving the selected object as a class
- Running the Class Browser and automatically displaying the class the selected object is based on
- Displaying a menu of registered add-ins, similar to the add-ins feature of the Class Browser
- Launching another builder
- Toggling the AlwaysOnTop property of the builder form
- Displaying the contents of the BUILDERB.TXT help file

Another interesting feature of BuilderB builders is that they are non-modal; while the builder form is open, you can click back on the Form or Class Designer window the object is in. Any changes you make to properties of the object are reflected back in the builder (a timer is used to handle this). You can turn this behavior off by setting the lAutoRefresh property of any control in the builder to .F.

Other notes about BuilderB:

- BuilderB includes a program called, strangely enough, BUILDERB.PRG. This program simply appends the directory it’s in to the VFP path so BuilderB files can easily be found. You might want to put your builder class library (such as SFBLDRS.VCX) into the same directory as BuilderB so pathing isn’t an issue.
- You can use builders created with BuilderB to help you create BuilderB builders (is your head starting to hurt yet? <g>). After you drop a BuilderB control on a builder you’re creating, you can right-click on the control, select Builder from the shortcut menu, and use a special builder to set the properties of the control, such as cProperty, lAutoRefresh, and lUpdateOnChange.
- BuilderB supports multiple levels of builders. You can add a custom BuilderX property to one of your classes and enter the name and class library of the BuilderB builder for that class. This overrides any builder name entered into the custom Builder property; choosing Builder from the shortcut menu launches the BuilderX builder rather than the one specified by the Builder property. However, you can still launch the builder specified in the Builder property by clicking on a button in the builder form. This means you could have one builder for a certain class (entered into Builder) and a specialized builder for a subclass of that class (entered into BuilderX). Launching a builder for the subclass would give your specialized builder, but you could then access the more general builder from the builder form.
- In addition to builders invoked from the shortcut menu, BuilderB allows you to create “drag toolbar” builders, which allow you to drag properties (such as font settings) and drop them on objects. BUILDERS.VCX includes several examples of these builders.
- Like other builders, BuilderB builders can do more than simply change some properties. For example, you could use the WriteMethod method to insert code into a method of the object.

## Summary

Builders can greatly increase your productivity, especially when used with containers that you would normally have to drill down into to change properties. With Ken Levy's BuilderB, you can literally create a new builder in a matter of minutes without having to know anything about how builders work. In future articles, we'll provide builders for complex reusable tools we create to make it easier to work with these tools.

*Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America, and spoke at the 1997 Microsoft FoxPro Developers Conference. He is a Microsoft Most Valuable Professional (MVP). 75156.2326@compuserve.com or dhennig@stonefield.com.*

*Yuanitta Mohart is a software developer with Stonefield Systems Group Inc. She has been developing applications in FoxPro 2.x and VFP for over six years. ymorhart@stonefield.com.*