Building a Builder Builder

Doug Hennig

Creating builders is a snap using enhancements to Ken Levy's BuilderD technology described in this month's article.

Last month, we looked at BuilderD, a new data-driven builder technology created by Ken Levy. BuilderD allows you to create a builder for a class by simply adding records to a builder definition table (BUILDERD.DBF, located in the WIZARDS subdirectory of the VFP home directory).

As easy as it is to create a builder using BuilderD, wouldn't it be even better if there was a visual front-end to BUILDERD.DBF, along the same lines as the forms we provide to our users? What I'm talking about is something that builds builders; yes folks, a builder builder. In fact, we'll use BuilderD itself to build the builder builder (does that make it a builder builder builder?).

As an aside, about a decade ago, I worked for a company that had a pretty cool communications setup. We had a communications server (fast modems were pretty expensive back then) on our LAN that had the remote control software pcAnywhere installed on it. We had a program called LANAssist (essentially a LAN version of what pcAnywhere does) that allowed me to control that server from my workstation, so I could fire up pcAnywhere on it, and have it dial into a client's communication server. I would then use LANAssist on that server to take over any user's workstation on the client's LAN. What got really hard to follow was: when I type DIR, am I getting a directory of my machine, our communication server, the client's communication server, or the client workstation I was controlling? It took serious concentration to remember what machine your keyboard was actually affecting. I think you'll find this analogous to what we'll be doing here: "is this builder managing my object or is it managing the builder that manages my object?" Consider yourself warned <g>.

The source code available from the Subscriber Downloads site for this month includes SFBUILDERS.VCX, which contains subclasses of BuilderD classes (which are contained in BUILDERD.VCX in the WIZARDS subdirectory of the VFP home directory) that provide the functionality we need. There isn't enough room here to go over all the code in these subclasses, so I'll hit the high points.

First, a couple of definitions: "target object" refers to the object that you brought the builder up for, and "property control" refers to a control in the builder that maintains the value of a property of the target object.

SFBuilderBuilderForm

The first class we'll look at it is SFBuilderBuilderForm. This class, subclassed from BuilderDForm (the builder form class for BuilderD), is the builder builder, and what you'll specify in the BuilderX property of a class (you'll enter "<directory>\SFBuilders, SFBuilderBuilderForm"). This subclass has some additional buttons added:

- An "add property control" button that adds a new property control to the builder and displays a builder for that property control so you can specify what property the control is bound to, what caption to use, and its size and position.
- An "edit builder caption" button that brings up a builder so you can change the caption for this builder.
- A "save" button that updates the builder definition records in BUILDERD.DBF for the builder we're working on.
- An "export" button that writes the builder definition records to another table. This allows you to ship this table to someone else, and they simply have to APPEND FROM it into BUILDERD.DBF to get a copy of the builder you created.

The Load method of SFBuilderBuilderForm changes the cProgramPath property to point to the WIZARDS subdirectory of the VFP home directory. This is necessary because the normal behavior of Load (which is first executed using DODEFAULT) is to set this property to the directory this class' class library is located in. Since we may not want to install SFBUILDERS.VCX in the WIZARDS directory but we still need to access files in that directory, this is a necessary step.

The Init of SFBuilderBuilderForm uses DODEFAULT() to do the normal behavior, then calls a custom method called CreateBuilderRecords. This method adds records to BUILDERD.DBF (if they don't already exist) that define the builders used to maintain our builder. Init then ensures that the correct number of pages is displayed and that any custom properties the builder manages but don't exist in the target object are added to the target object (more about this later).

The RightClick method of the form class calls a custom ShowMenu method. This uses the same mechanism I discussed in the February 1999 column on the FFC_ShortcutMenu class; RightClick calls ShowMenu, which instantiates a _ShortcutMenu object and calls the ShortcutMenu method to populate the object, then activates the menu. This provides us with context menus for both the form and for property controls (so we can, for example, right-click on a control and select a function to edit or remove it).

Both the "add property control" button's Click method and the "Add property control" function in the context menu call the form's AddPropertyControl method to add a new property control. This method checks how many controls are on the last page of the pageframe and adds a new page if necessary (the nMaxObjects property of the oBuilderDB object on the form defines how many controls we'll put on a page). It then adds BuilderLabel (for the caption of the property) and BuilderTextBox objects to the page and binds the text box to the Tag property of the target object by setting its cProperty property to "Tag" (I had to pick *some* property, and figured that something as global as Tag would be the best choice). Note that the BuilderTextBox class it adds doesn't come from BUILDERD.VCX, but is instead the BuilderTextBox class in SFBUILDERS.VCX (which, as you may expect, is subclassed from the BuilderD version). The reason for using a subclass is that I've added code to the RightClick method of my subclass so we can have a context menu for property controls (there's a subclass of BuilderCheckBox in SFBUILDERS.VCX for the same reason). Why did I name these classes the same as their parent classes and not "SFsomething"? That's because when an existing builder definition is loaded from BUILDERD.DBF, oBuilderDB.AddObjects creates BuilderTextBox and BuilderCheckBox objects as the property controls. With SFBUILDERS.VCX earlier in the class search chain that BUILDERD.VCX (done with SET CLASSLIB TO SFBUILDERS, BUILDERD), I ensure that my subclasses are used instead of the BuilderD classes. Finally, AddPropertyControl calls the custom EditObject method, passing it a reference to the new control, to display a builder for the new property control. EditObject simply instantiates BuilderDForm (yes, we're using the regular BuilderD form for our property control builder), passing it a reference to the control, and tiles it below and to the right of the current builder form.

Both the "edit builder caption" button's Click method and the "Edit builder caption" function in the context menu call the form's EditBuilderCaption method. This method simply calls EditObject, passing a reference to the builder form, so it displays a builder for the form, which simply provides a way to edit the caption of the form.

The Save method is called from both the "save" button's Click method and the "Save" function in the context menu. This method spins through all the property controls in the form and creates or updates a record for that control in BUILDERD.DBF. The Export method (called from both the "export" button's Click method and the "Export" function in the context menu) also calls Save, but passes it the name of a table to export to which it obtained from you using GETFILE().

Several of the methods in SFBuilderBuilderForm aren't currently called from anywhere, but are there for future use. BuilderD doesn't currently have great support for which page a property control is placed, so methods I created to add new pages, remove pages, and change the caption of pages aren't currently used (you'll see commented out code in ShortcutMenu, which populates the context menu, that would provide functions calling these methods).

Right-clicking on a property control displays a context menu with functions to edit the control (it simply calls EditObject, passing it a reference to the control), remove the control (which calls RemovePropertyControl to remove the control and any associated label object), or reset the value of the property the control manages to its default value (by calling the DefaultReset method; the code for this method is in BuilderBaseForm).

Other methods in SFBuilderBuilderForm are just support methods. For example, FindCaption locates the label object that provides the caption for a property control by looking for the object with a TabIndex value of one less than the control's TabIndex value (pretty low-tech, I admit, but I didn't want to change any code in any BuilderD classes, so this was the only way I could think of). FindClass and FindProperty find a specific CLASS or PROPERTY record in BUILDERD.DBF.

If you recall from last month's article, the AddObjects method of the oBuilderDB object on the form is the data-driven engine of BuilderD; it reads builder definition records from the BUILDERD table and adds controls to the builder form. This method was overridden in SFBuilderBuilderForm to provide additional functionality: it adds a record for the class the builder is for to the BUILDERD table if one doesn't already exist and it adds SFBUILDERS.VCX before BUILDERD.VCX in the SET CLASSLIB command. The first change is needed because, otherwise, BuilderD would give you an error that there are no builders registered for this class; we want to auto-register a builder for the class the first time a builder is invoked for it. The second change ensures that our BuilderTextBox and BuilderCheckBox classes are used rather than BuilderD's versions, so we get a context menu when we right-click on a property control.

Other Builder Classes

As I mentioned earlier, the BuilderCheckBox and BuilderTextBox classes in SFBUILDERS.VCX are subclasses of the same named BuilderD classes. In addition to providing right-click behavior, these two classes have nTop and nLeft properties, with assign methods for each that set the custom lMoved property to .T. when these properties are changed. This allows us to detect when a property control was moved (the Left and Top controls in the property control builder are bound to nLeft and nTop rather than Left and Top directly). Only if a property control is moved do we store its Left and Top values to the BUILDERD table.

Another class in SFBUILDERS.VCX is SFPropertyCaption. This control is used to manage the caption for a property control. Why have a special class for that? Why not just use a normal BuilderTextBox object (which this is subclassed from)? The reason is that if the property control is a BuilderCheckBox, it has a Caption property, so the control manages that property. However, if the property control is a BuilderTextBox, it doesn't have a Caption property but instead has an associated label object, so the control must manage the Caption property of that object. Since BuilderD classes are intended to manage the properties of a single object, SFPropertyCaption had to override a few methods to allow it to handle this situation.

The SFBuilderPropertyComboBox class is a subclass of BuilderComboBox. It presents a list of all of the writable properties of the target object (obtained using AMEMBERS() to get a list of all properties and then checking PEMSTATUS() of each one to eliminate those that are read-only). Although its main goal is to change the cProperty property (which determines which property is being managed) of the property control it's managing (are you getting a headache thinking about these multiple levels of management?), it has a couple of interesting behaviors. First, if the property control is a BuilderTextBox but you've just changed the property it manages to a logical one (such as Enabled), it removes the BuilderTextBox and its associated label object, and puts a BuilderCheckBox in its place. It does the opposite if you change from a logical property to one of another type. Second, if you enter the name of a property that doesn't exist, you'll be prompted to create the property. If you agree, the builder uses AddProperty to add the new property to the target object. I don't really recommend doing this in an object dropped on a form, because this amounts to instance programming; if you really need a new property, you should consider using a subclass instead. However, this is a quick way to create a new property in a class *and* have the builder manage it in one step.

SFBuilderAddButton is a button class (subclassed from BuilderCommandButton) that's added to the builder form for a property control using the SFBuilderButtonLoader class (this scheme of having a loader class to add a button object to the builder form was discussed in last month's article). The Click method of SFBuilderAddButton simply calls the AddPropertyControl method of the builder form the property control is on (there's that multiple levels thing again) to create a new property control, then switches the current builder form to manage the new property control rather than the one you brought it up to manage in the first place. This means you can click on the Add Property Control button to add a new property control to the builder, and in the builder that comes up to manage it, create another one. This is a fast way to add multiple property controls in a hurry.

Trying it Out

Let's see SFBuilderBuilderForm in action; it's actually a lot easier to use than it was to describe. Create a subclass of the VFP TextBox class called MyTestText in TEST.VCX. Add a BuilderX property and set it to "SFBuilders,SFBuilderBuilderForm", then bring up the builder for the class. Notice that even though we didn't create any records in the BUILDERD table, we still got a builder form anyway (of course, there are no controls on the form, but we'll change that in a moment). This is because SFBuilderBuilderForm automatically created a CLASS record for the class in BUILDERD when it didn't find one.

Click on the Add Property Control button. You'll notice that a textbox and label appear on the builder form, but then another builder form appears on top of the original builder form. This new form is the builder for the property control we just added. In the Property combobox, select "Tooltiptext" and change the Caption to "Tool Tip Text" and the Width to 250. Move this builder form aside and notice that the property control on the original builder form has been changed accordingly. Click on the Add Another Property button in the second builder form and notice that another property control was added to the original builder form and this builder form now maintains it. Select "Statusbartext" for the Property and change the Caption to "Status Bar Text" and the Width to 250. Add another property control and select "Readonly" for the Property and change the Caption to "Read-Only"; this time, notice the property control in the original builder changes to a checkbox. Close the new builder form. Figure 1 shows the builder we've built and Figure 2 shows the property control builder.

Figure 1. The BuilderD builder we've created for MyTestText objects.

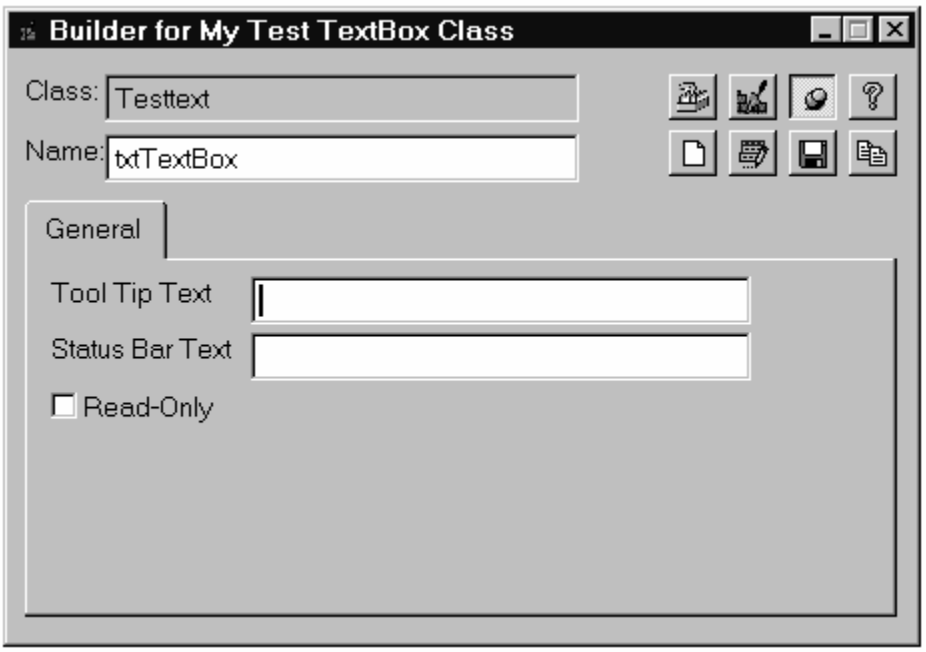
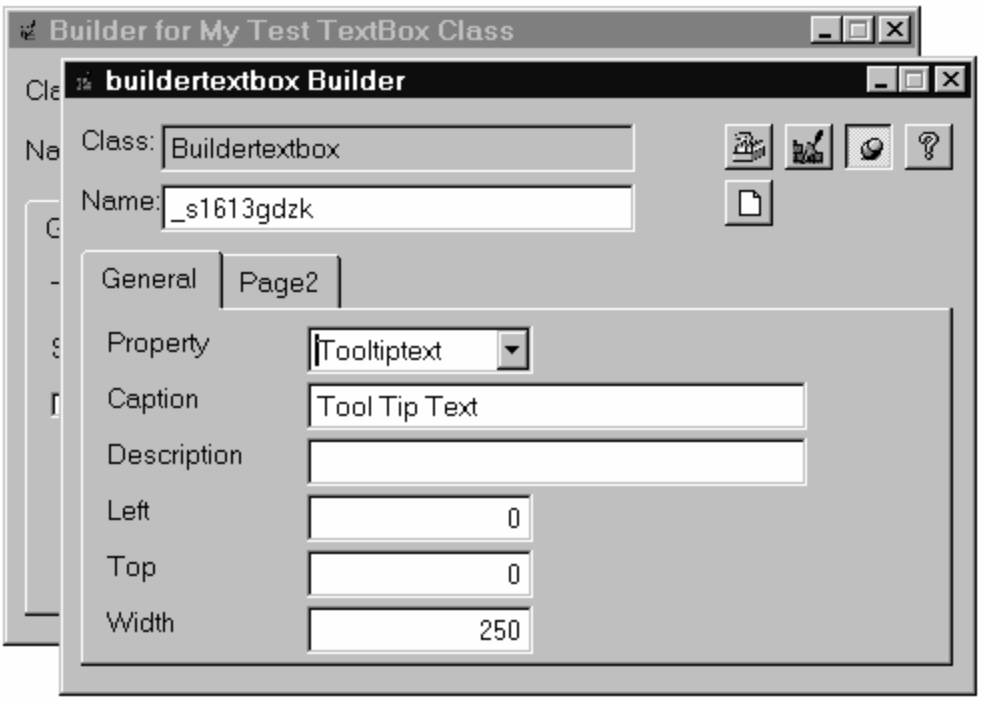


Figure 2. The property control builder.



To edit one of the property controls, right-click on it and select Edit Property Control from the context menu; the same property control builder you saw a moment ago appears. To remove the property control, choose Remove Property Control from the menu. To reset the value of this property in the target object to the default value, choose Reset to Default.

Let's change the Caption of the builder form to something more suitable. Click on the Edit Builder Caption button and enter "Builder for My Test Textbox Class" for the Caption in the SFBuilderBuilder Builder form that appears. Close this second builder.

If you close the builder without saving, the next time you bring up the builder for any MyTestText object, you'll have a builder with no properties again. To save the builder definition in the BUILDERD table, click on the Save Builder button. If you want to export the builder definition to another table, click on the Export Builder button and enter a filename in the dialog that appears. You can then send this table to someone else so they can import it into their BUILDERD table and have access to the builder you created.

Conclusion

BuilderD is a great tool for creating builders for your classes, and the enhancements I described in this article make it even better. Thanks to Ken Levy and Steven Black for reviewing my enhancements and suggesting improvements.

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 and Stonefield Query. 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 the 1997 and 1998 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). He can be reached at dhennig@stonefield.com.