April, 2001

Have it your way!

Add-ins provide an easy way to enhance the Class Browser

I don't use the Class Browser as much some people do. I like it for examining class libraries I didn't write and for turning a class into code that I can publish, but it's not a tool I use all the time. But recently, I needed to do something and thought the Class Browser should be able to do it for me. That was the beginning of my odyssey into creating a Class Browser add-in.

The task in question was a simple one, something we've all probably done a number of times. I was starting a new project and wanted to subclass every class in my base class library, putting the subclasses in a new project-specific library.

The Class Browser has a button to create a new class from an existing class. It brings up the dialog in Figure 1, where the destination defaults to the current library. To use that button to subclass the whole library, I'd have to select each class, click the button, then fill in the name for the class and specify the destination library. Definitely too much work.

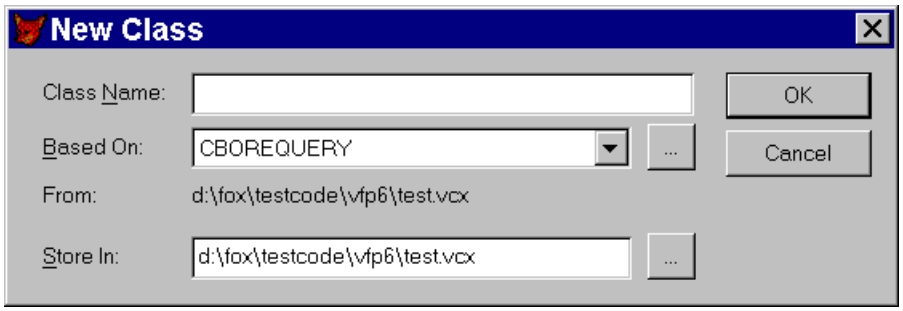

Figure 1. The New Class dialog – the New Class button in the Class Browser brings up this dialog, making it tedious to use for subclassing a whole library.

But this still seemed like a Class Browser task to me. I realized it was time for me to write my first Class Browser add-in.

What's an add-in?

Like many of VFP's tools, the Class Browser (I'll often refer to it as "the Browser" in this article) has an open architecture. The Browser is an object, with many public methods. One way to add functionality to the Browser, then, is to subclass it and either add methods or override

existing methods. But one look at the Class Browser's code convinced me that I didn't want to do that.

The Class Browser also offers an extension method that doesn't require you to dig so deeply into its guts. An add-in is a program attached to the Browser that can manipulate the open objects, using the Browser's methods and properties. You can attach an add-in to any Class Browser event or simply put it on a menu of add-ins. I chose to do the latter for my library subclassing add-in.

Attaching an add-in

Connecting an add-in to the Class Browser is simple. You call the Browser's AddIn method, passing appropriate information and the add-in is inserted into the Browser's registration table (Browser.DBF in VFP's Home directory). You only have to register an add-in once and you can use it thereafter. To get to the AddIn method, you can use the public _oBrowser reference that's created whenever you open the Browser.

The syntax for registering an add-in is:

```
_oBrowser.AddIn( cName , cProgram [, cMethod ])
```

cName is the name for the add-in-when you put an add-in on the menu, cName is the name that appears. cProgram is the program containing the add-in code, including its path. cMethod lets you specify that the add-in should be fired by a particular Class Browser method. If you omit this parameter, the add-in appears on the Add-ins menu. There are a couple of other optional parameters, but you're not likely to use them for most add-ins.

You can unregister an add-in by specifying only cName and passing .null. for the second parameter:

```
_oBrowser.AddIn( cName, .null. )
```

Using an add-in from the menu

To run an add-in that you haven't hooked to an event, right-click on either of the lower panes (the description panes) or on an unused section of the toolbar row. The Browser's context menu appears. Choose Add-ins and a menu of add-ins appears. Figure 2 shows the Class Browser with the Add-ins menu open – it contains only my new add-in.

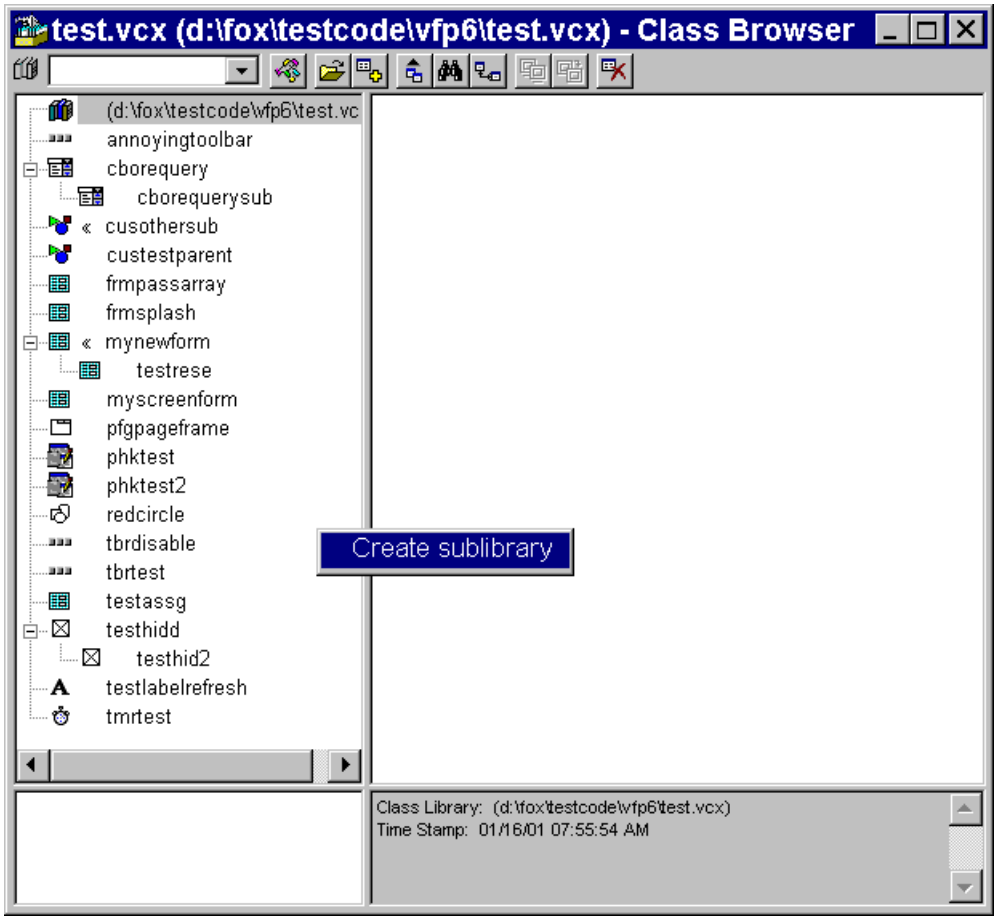

Figure 2. Running an add-in – To use an add-in, you right-click on one of the description panes or an unused section of the toolbar row. Choose Add-ins from the menu and the Add-ins menu appears. Choose the one you want.

Creating an add-in

Now that we know how to register and use an add-in, we reach the harder question. How do you write one? A key element is that every add-in receives, as a parameter, an object reference to the Class Browser object. That makes the Browser's properties and methods available without having to use the _oBrowser global reference. (That's particularly important if you might have more than one instance of the Browser open. In that case, you don't know which instance of the Browser _oBrowser refers to.)

The Browser object has a number of collections. Two of them are particularly useful for writing add-ins. aFiles contains a list of the files (class libraries, forms, etc.) open in the Browser, with nFileCount holding the number of files. aClassList contains a list of all the classes open in the Browser; nClassCount holds the number of classes.

For my add-in, I needed to work with only the aClassList collection. In particular, I used the first column containing the name of the class and the sixth column holding the name of the containing class library. (All 9 columns of aClassList are documented in the "Class Browser Properties" Help topic.)

Some other Class Browser properties are useful here, too. The cClass property contains the name of the currently selected class, while cClassLibrary contains the name of the class library containing the selected class. If the highlight is on a class library name, rather than a class name, cClassLibrary contains that library name and cClass is empty.

The Browser has lots of exposed methods. My first plan for my add-in was to use the NewClass method. However, it turned out that the method wouldn't allow me to store the new class in a different library than the class it was based on.

So, I realized that I'd need to create the class with code. Fortunately, the CREATE CLASS command can be used programmatically. Unfortunately, it doesn't include a NOSHOW clause to create the class without opening the Class Designer. That meant that I needed a way to close the Class Designer for each class. A little experimentation indicated that KEYBOARDing the sequence {CTRL+F4}Y would do the trick. ("{CTRL+F4}" closes the Class Designer window. "Y" answers the prompt as to whether the class should be saved.)

First attempt

My first version of this add-in was quite simple. It did a lot of checking to make sure that there was something to do. Once it made sure that the Class Browser was open, that a class library was open, and so forth, it prompted for the location of the new class library. After that, it looped through the aClassList collection, and for each class in the appropriate class library (because you can have multiple class libraries open in the Browser), issued a KEYBOARD command and CREATE CLASS command. Here's the main processing loop of that version:

```
WITH oBrowser
   FOR nClass = 1 TO .nClassCount
      * Only the ones in the current library
      IF .aClassList[nClass, 6] = .cFileName AND ;
         STRTRAN(STRTRAN(.aClassList[nClass, 1], ;
            "(", ""), ")", "") <> .aClassList[ nClass, 6 ]

            * Set up to shut down the Class Designer
            KEYBOARD "{CTRL+F4}Y"
```

```
            * Create the class
          CREATE CLASS (.aClassList[ nClass, 1] ) ;
             OF (cNewClassLib) ;
             AS (.aClassList[ nClass, 1]) ;
             FROM (.aClassList[nClass, 6])

       ENDIF
    ENDFOR
ENDWITH
```

This version had only a minor user interface – a call to PUTFILE() to get a name for the destination library. You'll find this program on this month's Professional Resource CD as NewLibOriginal.PRG.

Making it pretty

I quickly realized that there were a couple of things missing from my add-in. First, what should happen if some of the classes to be copied already existed in the destination class library? I wanted the ability to control the results - to indicate for each class whether to keep the existing class or overwrite it with a new subclass. This was significant not just to avoid overwriting existing classes but also because attempting to create a class with the same name as an existing class causes a prompt to appear. This prompt was interfering with the KEYBOARD command, requiring user action.

Second, I wanted a way to change the prefix and suffix of a class while subclassing. Many developers use a prefix or suffix to indicate the level of a class in their class hierarchy. For example, "a" might indicate "abstract" while "c" is for "concrete". Some use a prefix or suffix to indicate the client or project the class belongs to. So some method of specifying a prefix or suffix to be removed and a new prefix or suffix to be applied was called for.

What all this meant is that the add-in needed a user interface, not just a call to PUTFILE() to specify the destination library. From the user's perspective, two forms were needed: one to specify the destination library and handle the prefix and suffix information (figure 2) and a second for the case where the destination library already contains some of the classes (figure 3).

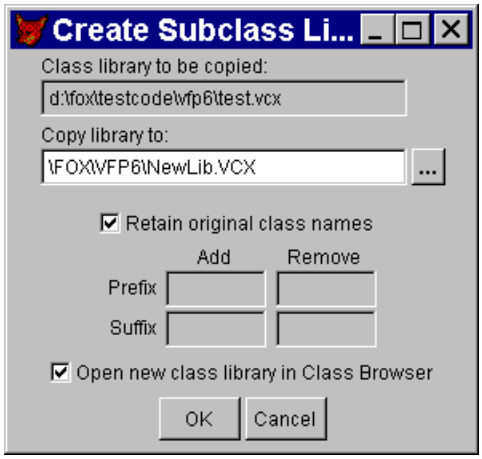

Figure 3. Specifying subclassing parameters–This form (actually a page) lets the add-in's user specify the destination library, determine how to handle prefixes and suffixes, and indicate whether the new class library should be opened in the Class Browser after subclassing is done.

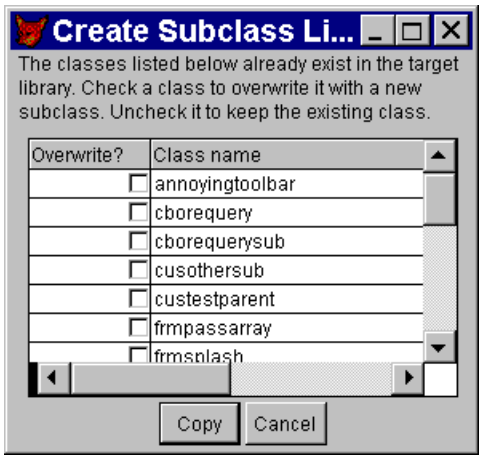

Figure 4. Handling duplicates–This form (actually a page) appears if the destination class library already contains any classes with the same names as classes to be subclasses.

It was at this point that Jim Booth jumped in to help. He took on the task of handling prefixes and suffixes while I worked on dealing with duplicates. Jim suggested that rather than creating two separate forms, we use a single form with a tabless pageframe. Each needed "form" could really be a page.

We both agreed that, while we would use the Form Designer to set up, tweak and test the interface for the add-in, once we got it working, we'd save the form as code and include it in the PRG file for the add-in. That way, all that needed to be distributed to others was a single file.

It was also clear to us that while we would normally use our own classes, for a tool like this, it made more sense to work with the VFP base classes so that we wouldn't need to distribute any class libraries with the add-in.

We migrated all the code that actually does the work into methods of the form or of the objects on the form. When we put all our work together, the main program of the add-in was significantly simplified. After doing all the checking (the same as in the original version), it just instantiates and runs the form:

```
LPARAMETERS oBrowser
LOCAL nClass, lHasClass, loFrmCopyClass
* First, make sure the Browser is out there
IF VarType(oBrowser) <> "O"
   WAIT WINDOW ;
    "Start the Class Browser before running this program.";
    NOWAIT
   RETURN .F.
ENDIF
WITH oBrowser
   * Now make sure there's a class library to work with
   IF .nClassCount = 0
      WAIT WINDOW ;
       "Open a class library in the Browser " + ;
       "before running this program." ;
       NOWAIT
      RETURN .F.
   ENDIF
   * Make sure it's a classlib, not a form
   IF UPPER(JustExt(.cFileName)) <> "VCX"
      WAIT WINDOW ;
       "This program only works with class libraries." ;
       NOWAIT
      RETURN .F.
   ENDIF
   * Make sure this classlib has classes
   lHasClass = .F.
   FOR nClass = 1 TO .nClassCount
      IF .aClassList[nClass, 6] = .cFileName AND ;
         STRTRAN(STRTRAN(.aClassList[nClass, 1], ;
            "(", ""), ")", "") <> .aClassList[ nClass, 6 ]
         * Found one
         lHasClass = .T.
         EXIT
      ENDIF
   ENDFOR

   IF NOT lHasClass
      WAIT WINDOW ;
       "The source library must contain " + ;
       "at least one class";
       NOWAIT
```

```
      RETURN .F.
   ENDIF
ENDWITH
* Display form here
loForm = CreateObject("frmCopyClassLib",oBrowser)
loForm.Show(1)

RETURN
```

The code to actually create the subclasses changed, too. The new version uses a couple of cursors to hold information about the classes in the source library and in the destination library. The various methods of the form work on these cursors rather than working directly with the Browser's collections. The Classes cursor contains a list of classes in the source library. Two logical fields, Preexists and CopyOver, indicate whether there's already a class with this name in the destination library and whether a new subclass should overwrite the existing class. Here's the main copying code from the CopyLibrary method of the form:

```
SCAN
   IF NOT Classes.Preexists OR ;
       (Classes.Preexists AND Classes.CopyOver)
      * If the class is new to the target or
      * it is marked to be an over write
      IF Classes.Preexists
         * If this is an over write
         *remove the existing class from the target
         REMOVE CLASS (ALLTRIM(Classes.NewClassName)) ;
            OF (lcNewLibrary)
      ENDIF
      * Set up to shut down the Class Designer
      KEYBOARD "{CTRL+F4}Y"
      * Create the class
      CREATE CLASS (ALLTRIM(Classes.NewClassName)) ;
         OF (lcNewLibrary) ;
         AS (ALLTRIM(Classes.OldClassName)) ;
         FROM (lcOldLibrary)
   ENDIF
ENDSCAN
```

Other methods of the form create and populate the two cursors, change the class names based on the prefix and suffix specifications, and handle all the various interface chores. You'll find the complete specification for this add-in on this month's Professional Resource CD and at Advisor.COM.

Learning more about the Class Browser

There are two good approaches for getting more information about what's available in the Class Browser. The Class Browser's exposed properties and methods are documented in the Help topics "Class Browser Properties" and "Class Browser Methods."

The other technique is to open the Class Browser and examine it using the Debugger. Open a couple of class libraries (and maybe even a form) in the Browser, then examine the properties of the _oBrowser object in the Locals window. This approach, in particular, helped me to see what was really going on.

In addition, there are two articles about the Class Browser available at the Microsoft Visual FoxPro website. The first (http://msdn.microsoft.com/library/default.asp?URL=/library/backgrnd/html/SB_Browser.htm) covers the Browser in general. The second (http://msdn.microsoft.com/library/default.asp?URL=/library/backgrnd/html/msdn_addins.htm) is specifically about creating add-ins.

Go for it

At first glance, adding behavior to the Class Browser add-in seems to be something of a daunting task. But with the add-in capability and the properties that are exposed, it's not that hard to get the Browser to do what you want it to.