September, 2007

Advisor Answers

Changing all Forms in an Application

VFP 9/8/7

Q: I'm updating an old application. The app has dozens and dozens of forms and I want to add an icon to each form. Is there a quick way to do this, or do I have to modify each and every one of them individually?

–Sytze de Boer (via Foxite.COM)

A: The answer to your question depends on whether the forms are all based on the VFP form base class or on a custom subclass. To find out, open a form and check the Class property.

If it's a subclass (anything other than "Form"), you can simply add the icon to that class and all the forms will inherit it.

Since you're asking the question, though, the chances are good that everything is based on the form base class. If so, you have two options. The first is to modify every form to have the icon you want; the second is to create a subclass of form, giving it the icon you want, and then modify all the forms to inherit from that subclass.

Overall, the second is a much better choice since it'll make it easier to make future changes that apply to all your forms. Unfortunately, whichever approach you choose will be tedious, since you'll have to touch every form.

The easiest way to do that is to write some code to do it. (Obviously, back everything up first). If all the forms are in a project, you can do something like this:

```
LPARAMETERS cProject, cFormClass, cFormClassLib
LOCAL oFile
MODIFY PROJECT (cProject) NOWAIT
FOR EACH oFile IN _VFP.ActiveProject.Files
   IF oFile.Type = "K"  && form
      * Open the form as a table
      USE (oFile.Name) ALIAS __CurForm
      * Find the record for the form itself
      LOCATE FOR UPPER(Class)="FORM"
      IF FOUND()
         REPLACE Class WITH cFormClass, ;
```

```
                ClassLoc WITH cFormClassLib ;
            IN __CurForm
      ENDIF
      USE IN __CurForm
   ENDIF
ENDFOR
```

This code takes advantage of the Project object created whenever you open a VFP project. It cycles through all the files in the project; if a file is a form (type="K"), it opens that form as a table, and changes the class and class library stored in the form record.

If the forms aren't in a project, you have to work with the files directly, but it's still pretty simple. In this case, use ADIR() to find the form files and handle them the same way. This version operates on files in the current directory:

```
LPARAMETERS cFormClass, cFormClassLib
LOCAL aFormFiles[1], nFile, nFileCount
nFileCount = ADIR(aFormFiles, "*.SCX")
FOR nFile = 1 TO m.nFileCount
  * Open the form as a table
  USE (aFormFiles[m.nFile, 1]) ALIAS __CurForm
  * Find the record for the form itself
  LOCATE FOR UPPER(Class)="FORM"
  IF FOUND()
     REPLACE Class WITH cFormClass, ;
             ClassLoc WITH cFormClassLib ;
        IN __CurForm
  ENDIF
  USE IN __CurForm
ENDFOR
```

Note that both versions change only forms that are currently derived from the base form class. That keeps you from accidentally changing the class for forms already derived from another class.

Both programs are available for download at the beginning of this article.

If you're not sure how to create a form class, check out Andy Kramek and Marcia Akins' article in the July 2007 issue.

Finally, if you don't want to create a form class, but just want to add the icons to the forms (although I don't recommend that approach), similar code will let you do so. Use the same looping structure, but modify the Properties memo of the form file to set the Icon property.

–Tamar

Know when to use Visual FoxPro Forms or Formsets

VFP 9/8/7

Q: When I need to have several forms appear together, is it better to use individual forms or a formset?

A: Almost always, free-standing forms are a better choice than formsets. There are quite a few reasons, both technical and practical.

Let's begin with a little history. FoxPro 2.0 introduced the Screen Builder and with it, the notion of screen sets, a group of screens (what we now call forms) that would be displayed together. Personally, I loved screen sets because they made it possible to have a number of forms handled under a single READ. That is, users could cycle through the fields of all the screens.

When VFP was introduced, one of the design goals was to make it possible to bring FoxPro 2.x applications forward. To enable that goal, formsets were included in the object hierarchy. They provided a direct mapping from FoxPro 2.x's screen sets, and in fact, the Converter tool that comes with VFP turns all screens, not just multi-form screen sets, into VFP formsets.

But in VFP's event model, the elimination of READ also eliminated the key reason for using screen sets. Working with non-modal forms, every form is equally accessible, and there was no need to group them just to make them all available at once.

So the first reason to use forms rather than formsets is that formsets don't offer anything you can't get with forms. (That's actually not quite true; more on that later.)

The second reason to avoid formsets is a practical one. Most VFP experts (I actually suspect it's all VFP experts) recommend against them, and very few developers are using them. More importantly, very few of the people who beta test VFP use formsets. That means formsets get much less testing than forms, and thus are more likely to have bugs. In addition, if you do run into a problem, you're less likely to be able to find someone who has run into that problem and can help.

Third, formsets are a weird kind of hybrid object. They have some of the properties, events, and methods of forms, but not all of them. It can be unclear where to set a given property, and forms in a formset may not behave exactly like independent forms.

Fourth, with a formset, you have to load all the forms at once, rather than handling each when it's first needed. That can add a serious delay for the user.

Fifth, once you put a form into a formset, there's no good way to get it out as a form. The only solution is to save it as a form class rather than as a form. So, when you turn out to need one of the forms in the formset somewhere else in the application, your options are to load the entire formset and hide the forms you don't need, or to duplicate the one form as a form class. Loading the entire formset has a clear performance penalty. Saving the form as a form class has two negatives. First, anytime you duplicate something in an application, you're introducing a maintenance burden. Second, in an application that works with SCX-based forms and formsets, having one form as a class is another maintenance burden.

So, what benefits do formsets offer? Only two that I can think of. (Thanks to Christof for pointing these out to me.)

First, if you need to make things modal and want several forms accessible, a formset is the only way to go. That said, I've never run into that situation.

Second, you can bundle toolbars into formsets, so that you can have a form and a formset appear together. While this is convenient, it doesn't take much code to simply instantiate the necessary toolbar with a form, and doing so offers a lot more flexibility than a formset.

Overall, the negatives of formsets far outweigh their benefits, and I can't recommend using them for anything other than the multiple modal form situation. If you want some evidence that the VFP team doesn't recommend using formsets, take a look at IntelliSense and the Toolbox. Formsets aren't included in either. When you type CREATE in the Command Window, the list of choices doesn't include Formset. Similarly, the list of base classes in the Toolbox doesn't show formset, either.

-- Tamar

De-Cluttering the Visual FoxPro Toolbox

VFP 9/8

Q: I like the Toolbox better than the Form Controls toolbar for adding controls to my forms. I've added lots of categories and classes, but now there's so much there that finding what I need is a problem. Is

there any way to cut the Toolbox down to just want I need at any time?

A: I like the Toolbox a lot, too, and I know exactly what you're talking about. There are several things you can do to make the Toolbox work better for you. First, if you're working on a number of projects that use different class libraries, filters may be just what you need. The Toolbox lets you define filters that contain a subset of the categories and makes it easy to switch between filters.

To set up filters, right-click on the Toolbox and choose Customize. In the left pane of the Customize Toolbox dialog that opens, click on Filters. You'll see a page that looks like figure 1. The left column of the right pane (Defined Filters) shows all the filters you've set up, while the right column shows which categories are included in this filter.

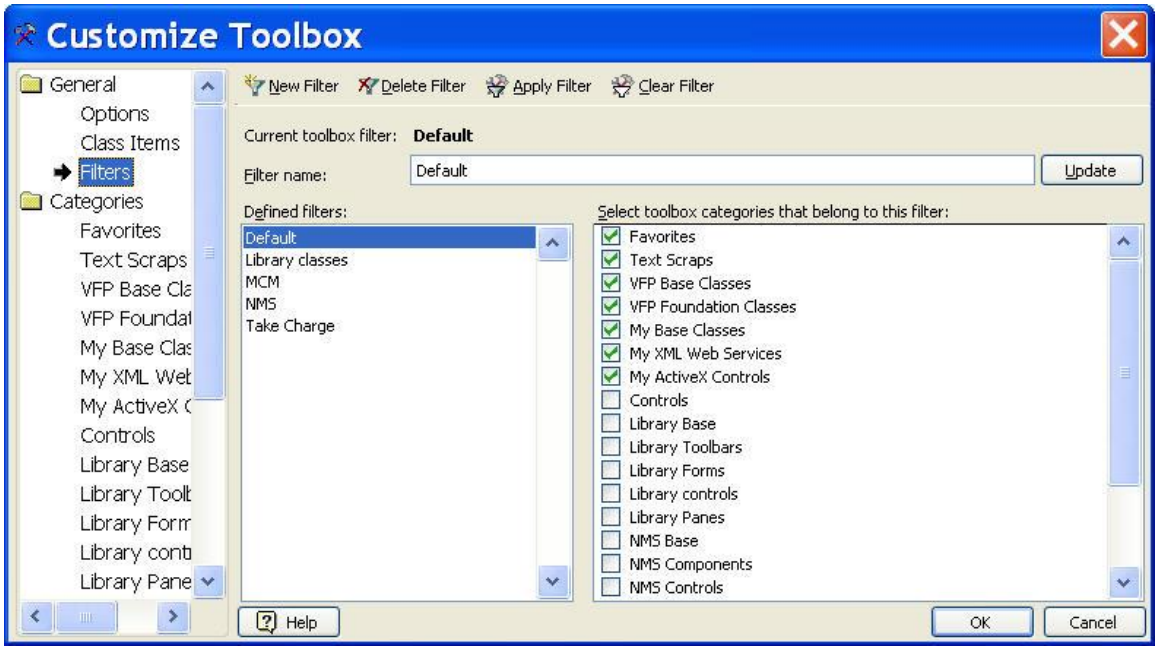

Figure 1: Filtering the Toolbox -- The Filters page of the Customize Toolbox dialog lets you manage filters that limit what you see in the Toolbox.

As you can see, I have five filters defined. The Default filter includes the categories that come with the Toolbox. That makes it easy for me to restore the Toolbox to its original appearance (something I sometimes need to do for writing or demo purposes). The other filters correspond to projects I'm working on. You can see in the right column that I include project names in my category names, so figuring out which categories to include in a filter is easy.

To set up a filter, click the New Filter button at the top of the dialog. Then, type in the name you want for the filter. Click the Update button to have the name updated in the Defined Filters list. (If you skip this step, it'll be updated when you close the dialog.) Then, check the categories to include in the filter. Click OK when you're done setting up filters.

When you return to the Toolbox, you'll find a new item, Filter, on the shortcut menu. When you choose it, you get a submenu of all the filters you've defined (plus a "none" item to clear filters). Click one to use it.

I find filters based on projects a very effective way to keep the Toolbox to a reasonable size, and to prevent me from using the wrong classes in a project. However, there are a couple of other ways to manage Toolbox clutter.

Although you add entire class libraries to the Toolbox at one time, you don't have to make all the classes in each class library visible. Having individual control over classes in the Toolbox is especially valuable if you set up your class libraries vertically, with an abstract class and its subclasses in the same class library. You can hide the abstract class in the Toolbox, so you don't accidentally drop it onto a form or another class.

To hide a particular class, right-click it in the Toolbox and choose Properties. In the Properties dialog, check the Inactive checkbox at the bottom left, then click OK.

If you want to hide a bunch of classes in a particular category, there's a quicker way. Open the Customize Toolbox dialog and choose the category in the left pane. The right pane shows the list of classes in that category. Uncheck those you want to hide, as in Figure 2.

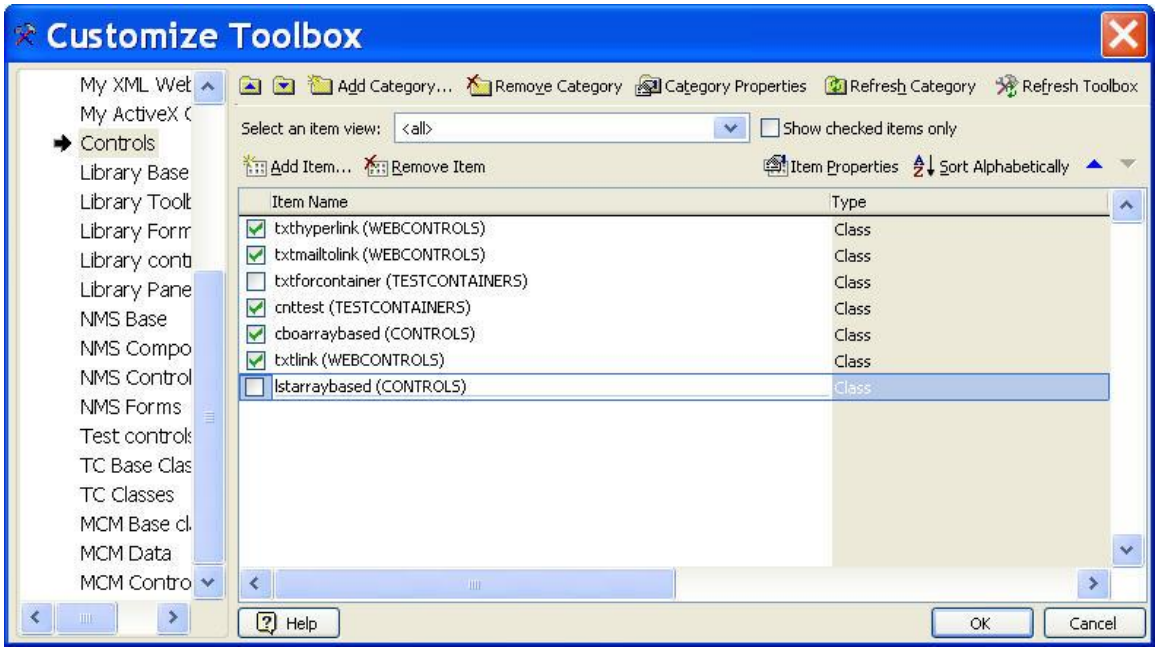

Figure 2: Hide Toolbox items -- You can hide some classes from a class library in the Toolbox by unchecking them in the Customize Toolbox dialog.

There's one other approach you can take to keeping the Toolbox manageable. As you'd expect, the Toolbox (which is written in VFP) stores its data in a table. You can maintain several Toolbox tables (perhaps one for each client or one for each project) and switch among them, as needed.

There are two ways to change the Toolbox table. To specify the table to use interactively, use the General page of the Customize Toolbox dialog. (Choose Options in the left pane.) Alternatively, you can change the Toolbox table with code. When you start the Toolbox, a reference to the Toolbox object is stored in _oToolbox. That object has a ToolboxTable property; set it to change the Toolbox table:

```
_oToolbox.ToolboxTable = "path\YourTable.DBF"
```

Whether you do it interactively or programmatically, having to remember to change tables when you switch between projects seems like an invitation to make mistakes and use the wrong classes. So I recommend using separate tables only if you're already running some code when you switch between projects. Then you can add changing the Toolbox table to that code. There are several possibilities for where that code might live. Among them are a project hook, the Environment Manager or a custom development menu.

It's said that in VFP, if you can do something at all, there are three ways to do it. It appears that de-cluttering the Toolbox fits right into that model.

-- Tamar