Collecting Objects

Doug Hennig

Collection classes can help you reduce the complexity of your applications by eliminating multi-column arrays or grouping related objects into one place. They can also help you create an interface to your applications that can be used by the outside world.

A collection is a group of related things. In Visual FoxPro, collections are everywhere: _SCREEN has a Forms collection, a form has a Controls collection, a pageframe has a Pages collection, and so on. Although they're implemented differently in different places, collections generally have some things in common:

- They have a property that returns the number of items in the collection. In the case of the Forms collection, this property is called FormCount; for Controls, it's called ControlCount; for Pages, it's called PageCount. For ActiveX collections, it's frequently called Count, making it easier to deal with in generic code. Sometimes this property is read-write (such as PageCount), but usually it's read-only.
- They have a way to add and remove items from the collection. With native VFP collections, the way varies; instantiating a new form automatically adds it to the Forms collection, while new pages are created by incrementing the PageCount property. With ActiveX collections, there are usually Add or AddItem and Remove or RemoveItem methods.
- They have a way to access items in the collection, usually by an index number. For the Forms collection, it's Forms[<index>]. For Pages, it's Pages[<index>]. ActiveX collections frequently have an Item method that can accept either an index number or the name of the desired item. Referencing an item in a collection by name is easier (and often requires less code) than by index number. Some VFP collections (like the Forms collection of _SCREEN) don't allow this, but some (like the Forms and Projects collections of _VFP) do.

OK, so what does that have to do with reusable tools? Well, I've been finding myself creating and using collections a lot lately. Often, these collections just consist of a class that contains an array of object references. Sometimes, the objects in a collection are very simple: they don't have any methods, but are just holders of properties. Other times, they're more complex objects, such as forms.

Here are some of the places where collections might be used:

- A menu wrapper class: a menu is a collection of pads, each of which is a collection of bars (actually, pads reference popups, but the wrapper class could hide this complexity). An object oriented menu would be much easier to work with than VFP's current menu system, which has changed little since FoxPro 2.0.
- A custom DataEnvironment class: VFP's DataEnvironment consists of two collections: cursors and relations. A substitute for the native DataEnvironment could provide more functionality, such as methods for saving and reverting tables rather than putting this code into a form.
- An application object's forms collection: unlike the Forms collection of _SCREEN, which just has an object reference to each open form, the forms collection of an application object could contain more useful information, such as which toolbars they reference, whether they support multiple instances or not, whether they add their captions to the Window menu, etc.
- VFP Automation server classes: one of the things that makes working with Automation servers like Word and Excel easy is their rich object models, which usually consist of many collections of similar objects. Many VFP developers are developing similar object models for their tools or applications, allowing them to be used in many more ways than just a VFP user interface.
- Replacements for arrays: I originally created my collection class to replace an array property with many columns. I found I was always forgetting which column certain information had to go into. It's a lot easier to understand and maintain code that simply sets or gets properties of objects in a collection than working with rows and columns of an array.

To make it easier to work with collections of objects, I created a couple of abstract classes intended to be subclassed into specific collection classes: SFCollection and SFCollectionOneClass.

SFCollection

The VFP 5 version of SFCollection (found in SFCOLLECTION.VCX) is a subclass of SFCustom, our Custom base class found in SFCTRLS.VCX. It has a Count property, which contains the number of items in the collection, and a protected aItems array that contains references to the items. The AddItem and RemoveItem methods add and remove items from the collection, and the Item method returns an object reference to the specified item in the collection. Why use a method to access items in the collection rather than just addressing the aItems array directly? Because we want to be able to reference items in the collection the same way ActiveX controls and newer VFP collections allow, either by index number or name. Here's an example:

```
oCollection.Item(5)
oCollection.Item('Customer')
```

VFP 5 doesn't allow an element in an array to be accessed by anything other than a numeric subscript, so we can't access aItems like this. Instead, we'll use an Item method that accepts either a numeric or character parameter. If the parameter is numeric, we assume it's the element number in the array. If it's character, we look for an item with that name.

The VFP 6 version of SFCollection is also based on SFCustom, but it does things a little differently. Because we now have access and assign methods, we can use an array called Item that contains the items in the collection and access this array directly.

The access method of a property is automatically fired whenever anything tries to access the value of the property. Similarly, the assign method is fired whenever anything tries to change the value of the property. Access and assign methods are named the same as the property with "_access" or "_assign" as a suffix; for the Item property, these methods are called Item_Access and Item_Assign. When you create a new property, check the "Access method" and "Assign method" checkboxes in the New Property dialog to create these methods. VFP automatically puts code into these methods to accept the necessary parameters and get or set the value of the property. However, you can override the behavior of these methods to do whatever you want. We'll see the code for some of these methods later.

The advantage of using this approach is that in the VFP 6 version, Item is a property not a method. This means you can use it to talk directly to an item in the collection. For example, you can use:

```
oCollection.Item[<index>].Property
```

In the VFP 5 version, you must use:

```
loObject = oCollection.Item[<index>]
loObject.Property
```

because Item is a method, not a property in this version. The VFP 6 version acts more like native collections in VFP and ActiveX controls. Another advantage of access and assign methods: Count isn't a protected property, so in the VFP 5 version, improper values could be stored to it. The VFP 6 version uses an Assign method to make it a read-only property.

The methods in the two versions of this class are almost identical; we'll look at the code for the VFP 6 version in this article. Here's the code for the AddItem method:

```
lparameters tcClass, ;
    tcLibrary, ;
    tcName
local lnCount, ;
    loItem
with This
    .lInternal = .T.
    lnCount = .Count + 1
    dimension .Item[lnCount], .aNames[lnCount]
    loItem = newobject(tcClass, tcLibrary)
```

```
    .Item[lnCount] = loItem
    if vartype(tcName) = 'C' and not empty(tcName)
        .aNames[lnCount] = tcName
        if not ' ' $ tcName and isalpha(tcName)
            loItem.Name = tcName
        endif not ' ' $ tcName ...
    endif vartype(tcName) = 'C' ...
    .lInternal = .F.
endwith
return loItem
```

This method accepts three parameters: the name of the class to create the new item from, the library that class is defined in, and the name to assign to the item. AddItem creates an object of the specified class (using either NEWOBJECT.PRG in VFP 5 or the new NEWOBJECT() function in VFP 6), stores the object in a new row in the items array (aItems for VFP 5 and Item for VFP 6), and stores the specified name so it can be used later to find the item (in the second column of the aItems array for VFP 5 and in the protected aNames array for VFP 6). If the name is a valid VFP name, the name is also stored in the Name property of the new object. An object reference to the new item is then returned so the code calling this method can set properties or call methods of the new item. Notice something interesting in the VFP 6 version of this code: the Count property isn't incremented. We'll see why this is unnecessary later. Also note the use of the lInternal property. This protected property is .T. only when method code of the SFCollection class is executing. Access and assign methods of a property fire even when that property's value is accessed or assigned by methods of the property's own class. However, you might want different actions depending on whether the property is accessed or assigned from inside or outside the class. This flag is used to distinguish internal calls from external ones. One last thing to note: this method uses the new VARTYPE() function, which is much faster than the TYPE() function we're used to using.

The RemoveItem method accepts the index or name of the item to remove as a parameter. If the specified item can be found (using the GetIndex method to find the element number of the item), the item is removed from the items and names arrays. As with AddItem, note the value of the Count property isn't changed here. Here's the code for this method:

```
lparameters tuIndex
local llReturn, ;
    lnIndex, ;
    lnCount
with This
    .lInternal = .T.
    lnIndex = .GetIndex(tuIndex)
    if lnIndex > 0
        adel(.Item, lnIndex)
        adel(.aNames, lnIndex)
        lnCount = .Count
        if lnCount = 0
            .InitArray()
        else
            dimension .Item[lnCount], .aNames[lnCount]
        endif lnCount = 0
        llReturn = .T.
    endif lnIndex > 0
    .lInternal = .F.
endwith
return llReturn
```

Item_Access (the Access method for the Item array property) in the VFP 6 version is identical in purpose to the Item method in the VFP 5 version, and has almost the same code:

```
lparameters tuIndex
local lnIndex, ;
    loReturn
with This
    lnIndex  = iif(.lInternal, tuIndex, ;
        .GetIndex(tuIndex))
    loReturn = iif(lnIndex = 0, .NULL., .Item[lnIndex])
```

```
endwith
return loReturn
```

This method accepts the index or name of the desired item as a parameter, and if that item exists, returns an object reference to it.

The GetIndex method is called from several other methods to convert a specified item index or name to the appropriate element number in the items array (or 0 if the item wasn't found). The VFP 5 and 6 versions are almost identical; here's the VFP 6 version code:

```
lparameters tuIndex
local lnCount, ;
    lnIndex, ;
    lcName, ;
    lnI
with This
    lnCount = .Count
    lnIndex = 0
    do case

* If the index was specified as a string, try to find
* a member object with that name.

        case vartype(tuIndex) = 'C'
            lcName = upper(alltrim(tuIndex))
            for lnI = 1 to lnCount
                if upper(.aNames[lnI]) == lcName
                    lnIndex = lnI
                    exit
                endif upper(.aNames[lnI]) == lcName
            next lnI

* If the index wasn't numeric, give an error.

        case not vartype(tuIndex) $ 'NIFBY'
            error cnERR_DATA_TYPE_MISMATCH

* If the index was numeric, give an error if it's
* out of range.

        otherwise
            lnIndex = int(tuIndex)
            if not between(lnIndex, 1, lnCount)
                error cnERR_INVALID_SUBSCRIPT
                lnIndex = 0
            endif not between(lnIndex, 1, lnCount)
    endcase
endwith
return lnIndex
```

Earlier, I noted that the VFP 6 version of AddItem and RemoveItem didn't adjust the value of the Count property. That's because Count_Access, the access method for this property, automatically calculates the value whenever Count is accessed (in fact, you'll see from this code that the value of Count is never in fact changed; the access method just returns the proper value). Here's the code:

```
local lnCount
with This
    lnCount = alen(.Item)
    lnCount = iif(lnCount = 1 and isnull(.Item[1]), ;
        0, lnCount)
endwith
return lnCount
```

Count is a read-only property: you can read its value but not assign a new one. In VFP 5, there was no way to do this directly with custom properties; you had to make the property protected and create a public method to return its value. In VFP 6, this is easily done: simply create an assign method for the property

that gives an error whenever something tries to change the value of the property (in this code, cnERR_PROPERTY_READ_ONLY is a constant defined in SFERRORS.H):

```
lparameters tuValue
error cnERR_PROPERTY_READ_ONLY
```

Of course, if your class needs to change the value of the property itself (remember in this case, Count is never changed but always calculated as required), you'll check the value of lInternal and store tuValue to the property if the flag is .T.

There are a few other methods in this class: Init calls the protected InitArray method to intialize the items array, and Destroy nukes all object references.

SFCollectionOneClass

The AddItem method of SFCollection requires the class and library for the object to be passed as parameters every time a new object is added to the collection. This can get to be drag when a collection consists of objects from the same class. SFCollectionOneClass is a subclass of SFCollection used for single-class collections. It has custom cItemClass and cItemLibrary properties that contain the class and library for the objects in the collection. Its AddItem method accepts just the name of the new object as a parameter, and uses DODEFAULT(This.cItemClass, This.cItemLibrary, tcName) to pass the class, library, and specified name to its superclass method.

Note that neither SFCollection nor SFCollectionOneClass will normally be used directly. Instead, you'll likely subclass SFCollectionOneClass for a specific collection and fill in the cItemClass and cItemLibrary methods for the objects this collection will contain. Also, I find it works best to use collaboration rather than subclassing to add functionality. In other words, don't create a subclass of SFCollectionOneClass and add a ton of features to this class. Instead, create a class with the desired functionality and associate it with an object instantiated from an SFCollectionOneClass subclass. For example, rather than having a fields collection (a collection of all the fields in the tables of an application) with all kinds of features, have a field manipulation object that has the features and applies them to its associated fields collection.

Samples

There are two sets of sample files, one for VFP 5 and one for VFP 6, in the ZIP file available from the Subscriber Downloads area. FILES.PRG creates a collection of file objects (from the File class in SAMPLE.VCX), then calls FILES.SCX to display the files in a listbox. Selecting one of the files causes the oFile property of the form to be set to the selected item in the collection, and several other controls in the form display properties of the item. The File class has no methods, just properties for the attributes of a file. You could easily add functionality to this class, such as the ability to change attributes (for example, setting the lReadOnly property to .T. would cause the file to become read-only), create or delete files, read from or write to files, copy or move files, etc.

FIELDS.PRG opens the VFP TESTDATA database and creates a collection of fields in the CUSTOMER table. The captions of these fields are displayed in a listbox using FIELDS.SCX. Selecting a caption in the listbox causes the appropriate field name to appear in the textbox on the form.

These are simple examples, but they are intended to show how SFCollection and SFCollectionOneClass can be used to create collections of objects. Next time you work on the design of a tool or application, consider using collections to replace complex arrays or expose the functionality of your application to the outside world.

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). CompuServe 75156,2326 or dhennig@stonefield.com.