Chapter 9
Forms

In many ways, the forms for an application are what it's all about—everything else is just along to hand 'em the saw! Forms allow the user to interact with the data, so from the user's standpoint, forms can arguably be considered the "heart" of the application. This chapter begins to look at forms as containers for the form controls that are discussed in chapters 10 and 11, and discusses issues that relate to forms without regard to what you put on them and what data they manipulate.

Loose coupling and forms

Without getting too esoteric, let me simply offer the advice that all forms *should* be capable of running independently from an application or from other forms. This accomplishes a couple of things. First, it greatly facilitates development, testing and debugging. I'm much more inclined to create a form incrementally (adding a little piece of functionality at a time and testing) if I don't have to fire up the entire application in order to test it.

Secondly, it encourages a good design principle known as *loose coupling* (more on this a little later). Clearly, it isn't possible to follow this stricture in all cases. At a minimum, some forms are designed to accept an optional parameter, and it will be necessary to test another form's ability to pass the argument and for the form being tested to receive and act on the parameter. Then, too, there are cases in which two forms operate cooperatively in some way. In general, if your app has a form that you can't simply run using DO FORM <form name> from the command window, the design of that form deserves a second look. The dependencies of such a form may indeed be unavoidable; however, forms that display this level of dependence should be the exception rather than the rule.

Steve McConnell, in his landmark book *Code Complete* (Microsoft Press, 1993), presents the concept of Loose Coupling. While much of his book does not address object-oriented or object-based systems specifically, the principles he sets forth are often just as applicable (with some conceptual adaptation) to an object-oriented development tool like Visual FoxPro. "Loose Coupling" in a procedural language simply means "black-boxing" your code. A loosely coupled routine can be called from anywhere in a system and it will do its job, without regard to where it is called or the state of the system when it is called. The knowledge that the rest of the system requires of the routine's inner workings is minimized to (at most) a parameter list.

If Steve McConnell's book Code Complete *(Microsoft Press, 1993, ISBN 1-55615-484-4) isn't on your bookshelf (or maybe even on your nightstand), it should be. In my opinion, any programmer or application developer absolutely* must *read this book if they consider themselves a professional. Other books are important, too (see Appendix 4), but this is the one you absolutely cannot ignore. If you can't spare the time or expense to find, buy and read this book, you might want to consider taking up something else for a living. I keep it on my desk (alongside* The Hacker's Guide to Visual FoxPro*) and read* *from it randomly on a regular basis for inspiration, just as others will consult the* Book of Changes*, or* The Prophet *or* Thoughts of Chairman Mao*.*

Extending this idea to Visual FoxPro forms, loose coupling means that our forms should ideally be able to run in any context. Clearly, a form is usually pretty dependent on the availability of data structured in a specific way, with certain expectations as to the names of tables, fields, indexes and so on. However, given that minimal requirement, the form should be able to function, at least in some manner, without dependence on other parts of the system.

In Chapter 3 I discussed "system-level services," and it is perfectly reasonable to have our forms make use of these services *when available*, but be able to function properly even if the system-level services in question are unavailable. As an example, consider an application with the following features:

- The application employs a form manager object. This object is responsible for actually launching all forms, keeping track of which forms are in use, and allowing us to programmatically manipulate our running forms—to cascade them, close them, tile them, and so on.
- The application is one in which one form frequently launches another related (modeless) form; thus it will make calls to the DoForm() method of the form manager object.
- The form manager object always passes a self-reference to any form it launches so that the form can easily communicate with the form manager.

As you might imagine, to support these features requires a method of the foundation form class called DoForm(). This method passes the name (and any optional parameters) to the form manager object, telling it what form it needs to launch, and has code in its Init() method to accept the reference to the form manager object. While at first it sounds like it will be difficult to test these forms without having the form manager object hanging about, the solution is to just bracket the necessary code as shown in **Listing 9-1**.

Listing 9-1. *Bracketing form method code to reduce dependence on system-level services.*

```
* Init() Method
LPARAMETERS toFormManager
ThisForm.oFormManger = toFormManager

* DoForm() Method
LPARAMETERS tcFormName, tuParm1, tuParm2
IF VARTYPE(ThisForm.oFormManager) = ""O"" ;
      AND NOT ISNULL(ThisForm.oFormManager)
   ThisForm.oFormManger.DoForm(tcFormName,tuParm1,tuParm2)
ELSE
   DO CASE
   CASE PCOUNT() = 1
      DO FORM (tcFormName) WITH .NULL.
   CASE PCOUNT() = 2
      DO FORM (tcFormName) WITH .NULL., tuParm1
   CASE PCOUNT() = 3
      DO FORM (tcFormName) WITH .NULL., tuParm1, tuParm2
   ENDCASE
```

```
ENDIF
```

The point here is that the form can make use of the system-level service if it's available, but can function just fine without it when necessary. There are several ways this can be accomplished.

First, as shown in Listing 9-1, the form (or any other object) can simply provide the required service itself, in the absence of another object that normally provides the service. The form is only interested in getting another form launched. The other services of the form manager object are not really of much interest to the form, although they are very important to the application as a whole. It's like a worker who needs to have a bunch of photocopies run off, but finds that the clerical person who normally handles this is out to lunch. The worker simply does the job himself.

Second, in the absence of an object to provide certain services, the form can instantiate the object that normally provides the services, or a substitute object that can provide the services. For example, an application object might provide a service to read information from and write information to the system registry. In the context of the application, the form, when run, normally asks the application object what 'its position and WindowState was the last time the user opened this form, so it can restore those settings. If the form is running from the command window (without the application object present), it could simply instantiate an object that provides those services (like the registry object that is included in the VFP 6 REGISTRY.VCX sample class library).

Finally, the form can simply exhibit certain default behaviors or settings in the absence of a system-level service. As in the example above, if the application object doesn't exist, the form simply centers itself on the screen at a default size.

You may feel that providing application-level support for a certain function, and then providing the means for a form to provide this service directly is redundant. It is, and should lead you to think about how you want to provide this functionality. I prefer to provide this kind of functionality globally to the application. You may prefer to simply have the form provide the service in all cases. With my approach, the ability of the form to provide the service is useful only during testing an action of a form that relies on the application-level service. Using the registry example above, my code checks for the existence of a registry interface object, and if it doesn't exist, simply establishes default values that it would have retrieved from the registry had the registry object been available. Does this mean that I can only test the code that depends on values obtained from the registry object by running the entire application? No. My registry classes "register" themselves as the registry object if they find that their parent object is a form. The following code is the Init() code from my registry class:

```
DODEFAULT()
IF VARTYPE(This.Parent) = "O" ;
        AND UPPER(This.Parent.BaseClass) = "FORM"
    llHasRegObjectProp = PEMSTATUS(This.Parent,"oregistry",5)
    IF llHasRegObjectProp
        ThisForm.oRegistry = This
    ENDIF
ENDIF
```

Thus, when I need to test the form's ability to read and write to the registry, rather than firing up the entire application, I can simply drop an instance of the registry object onto the form, run and test the form, then remove the registry object from the form.

If I were to find that I was testing this functionality on every form, I'd be inclined to include the registry object in my form foundation class, and abandon the approach of providing these services only through the an application-level registry object

However application independence is achieved for our forms, it makes our lives as developers much easier, and in the long run makes our forms and the applications in which they're used a little more flexible, and much better designed.

Private data sessions

Right on the heels of the transition from procedurally generated "screens" to object-based forms and the ability to create modeless forms without any special coding, the introduction of the *private data session* is one of the most significant advances in user-interface creation gained with Visual FoxPro. Only those of you who used FoxPro and other Xbase languages prior to Visual FoxPro 3.0 can appreciate this feature. Literally millions of lines of code have probably been written to allow different FoxPro 2.x screens to operate simultaneously on their own set of data without hosing the other screen's' data environment. The private data session "scopes" the tables, views, index files, relations, filters, record pointers, controlling indexes and many of the environmental settings that affect how data-related elements behave to the *form*. Without a private data session, forms would all be playing in the same sandbox. Form "B" could open a table, and if it wasn't careful to select an empty work area, could inadvertently close a table that was currently in use by Form "A". Form "B" or a procedure could change an environmental setting, say SET NEAR from "OFF" to "ON", significantly changing the behavior of Form "A".

Provided that your intention is to create a modeless, event-driven application, "2 - Private Data Session" should be just about the only setting that you use for the DataSession property. There may be situations in which you might have system-level tables open in the default data session and need to have a form to interact with them; however, this should be the exception.

Sharing data sessions

One issue that developers often have to face is the concept of a "child" form, that is, a form (usually modal) that is launched from and intended to interact with the data displayed on another form. One technique is to create a parent form with a private data session, and from that form launch a modal child form that uses a default data session. This will, indeed, allow the child form to interact with the parent form's data, and will, in effect, "share" the parent form's data session. However, there is a piece of anomalous behavior that has not (in my experience) caused any difficulty, but certainly causes many of us to think twice about using this method.

When a form with a private data session is run, its data session is identified by number and the name of the form that created it in the Data Session window (for example, frmTest1(2)). When this form launches a modal form that uses the default data session, the Data Session will show the new form name along with the data session (that is, frmTest2(2)). So far, so good. Now, however, when the second modal form is closed, the Data Session window shows Unknown(2). VFP knows what data session is in use, but for some reason no longer recognizes the form participating in this data session. If you want to use this method of sharing data sessions, be aware of this behavior, so that if something does get a little squirrelly at some point, you won't be too surprised. On the other hand, Microsoft has indicated that this behavior is indeed "by design," and that no unexpected or undesired effects of this anomalous reference to an "unknown" data session are to be expected.

Another issue with sharing a data session in this way is that if the child form is *not* modal, the parent form can be destroyed, taking the data session with it and closing all tables that the child table is using. Some steps must be taken to ensure that the parent form cannot be closed, leaving the child form hanging around. The easiest way to do this with a non-modal child form is to link the child form to the parent form via a property of the parent form, so that destruction of the parent form destroys the child form. Create a new property of the parent form, something like oChildForm, and then launch the child form with a line of code like this:

```
DO FORM <childForm> NAME ThisForm.oChildForm LINKED
```

Running the child form in this way, using the NAME…LINKED keywords, allows the parent form to call the methods and set the properties of the child form if necessary. It also allows the child form to be closed programmatically, and causes the child form to be destroyed automatically when closing the parent form.

I prefer to have all of my forms use a private data session, but have successfully used the "private parent, default child" technique described above. If I have two forms that must share a data session, another technique I use is to pass the parent form's DataSessionID to the child form as an argument, and the Init() code of the child form then sets its DataSessionID to match that of the parent. Just as with the technique of setting the child form's DataSession to "1- Default", the tables are opened and closed by the parent table, and thus can leave the child table without any tables to work with if the child table isn't modal. In addition to the technique illustrated above, setting the DataEnvironment.AutoCloseTables property to .F. in the parent form also prevents this problem. If the child form isn't running, the tables are closed anyway when the parent form's destruction also destroys the data session.

In general, I haven't found too many uses for FormSets, given the availability of pageframes, and now with the inclusion of scrollbars in VFP 6 forms, formsets in my opinion have even less utility. However, they are very well suited to sharing data sessions between different forms. The problem is that it isn't possible to get any form in a formset to behave as a modal child form—even if the formset is modal, all forms within the set are active, and the user can switch freely between the forms within the set.

Making the modal/modeless decision at runtime

While most of our forms are modeless, a modal form in the right place at the right time can really do the trick. For instance, if you have a "Search" or "Filter" form, it doesn't make much sense to allow the user to return focus to whatever form called the Search or Filter form until they've either made their choices or canceled the operation.

However, not all forms fall easily into either the modal or modeless category. If I have a search form that allows me to search and narrow down the list of all customers by name, sales volume, territory, zip code or last activity, and I call this form from the form that allows me to edit customer information, I want it to be modal. This forces the user to either select a customer, who'se ID is passed back to the calling form, or to cancel the search, in which case the user is returned to the calling form in its previous state. On the other hand, If I launch this search form from a menu pad that says "Customers", allowing the user to select a customer and then launching one of several customer-oriented forms based on that selection, I want the search form to be modeless. How can I make the form modal in one situation but modeless in another?

The WindowType property is indeed read/write at runtime, but take care when changing this property. When a modeless (WindowType = 0) form is run, any lines of code following the DO FORM command are executed after the form is instantiated. By contrast, when a modal form (WindowType = 1) is launched, the instantiation of that form introduces a *wait state*. This means that the execution of the code following the DO FORM command does *not* execute until the form is closed. If a modeless form is instantiated, and its WindowType property is changed from 0 to 1 by calling the Show() method (by passing an argument of "1"), the wait state commences with the execution of the Show() method. Thus you can instantiate a form, and then immediately call the form's Show method to both make the form visible and change its WindowType property:

```
DO FORM <FormName> NAME oForm NOSHOW
oForm.Show(1)
```

Doing this will introduce a wait state with the call to the Show() method. So far, so good.

However, sometimes it's necessary to have the form return a value using DO FORM <FormName> TO <memvar>. In this case, an error will occur as soon as the calling routine tries to make the form visible. Visual FoxPro expects the form to be modal when run with the TO <memvar> clause, and in this case the calling routine executes another line of code immediately after the form is instantiated.

The solution is to toggle the WindowType within the Init() or other method of the form (called by the Init()), but to do so by changing the property directly and *not* by calling the Show() method. Remember, Show(1) introduces a wait state that is terminated only with the destruction of the form. This puts us in a catch-22 situation: The form can't be terminated because one of its methods is executing. You've probably observed this when you suspended execution of a form method and then tried to release the form before canceling or resuming the method. The form will usually become invisible, but you can't edit the form in the form designer because the form hasn't been destroyed. Trying to force destruction of the form with CLEAR ALL will trigger a "Can't clear the object in use" error. The object is "in use" because code in one of its methods is still in the process of being executed (though it is currently suspended).

Thus, depending on the circumstance, the proper method of making an otherwise modeless form modal at runtime is to either use the Form.Show(1) method from the calling routine, or toggle the WindowType property in the form's Init() method.

Continuing with the example of the search form mentioned above, the calling form passes a self-reference to the search form, so that the search form knows that it should be run modally. If its Init() method does not see a form reference passed, it doesn't change the WindowType property, and it runs as a modeless form. This also has the advantage of allowing such a form, made modeless at runtime, to return a value to the calling procedure. This can't be done when setting the WindowType property to "1 - Modal" from the calling procedure—an error is triggered as soon as VFP detects (by recognizing that the next line in the calling procedure is executing) that you've executed a DO FORM…TO on a modeless form.

Passing data between forms

As with procedures and functions, values can be passed to a form by launching it with arguments that are received as parameters by the form's Init() method. As with procedures and functions, a form (provided it's modal) can return a value to the calling routine. However, some limitations to this process can present problems, particularly when multiple values need to be transferred to or retrieved from a form:

- Entire arrays can be passed to forms by reference, but go "out of scope" as soon as the Init() method code completes, necessitating transfer of the array data to a form array property.
- Object (including form) array properties cannot be passed as arguments, nor can they be returned as return values from the Unload() event of the form.

The solution to these problems is to stop thinking in the usual terms of "passing data." In procedural programming, data is passed from one routine to another as arguments/parameters, (dare I say it?) public or global memory variables, or as a value returned from a function call. A common practice in Visual FoxPro is to call a method of an object, or instantiate an object and pass data to it via an argument: DO FORM <whatever> WITH <some value>. If it's necessary for the form or the function to handle a set of data, an array is passed by reference, which has the added benefit (for a function, anyway) of allowing the function to return multiple values in the array.

Why be limited to this way of passing data when working with *objects*? After all, an object can contain a virtually unlimited amount of data (of any type) in its properties. Some folks have suggested creating a "wrapper" procedure that establishes a private array that is scoped to the procedure, and therefore to any form the procedure calls. Others have suggested passing an object containing an array property to a form instead of an array. The form can then store a reference to the passed object to a property and manipulate the array property; and the calling routine (which passed the reference to the object) can then retrieve the values from the array after the form runs. When working with forms, however, the "middle man" can be eliminated by simply passing a reference to the calling form, and allowing the called form to manipulate the calling form's properties directly.

Going back to the search form mentioned previously, one might execute a line like this in a customer editing form launching the search form:

```
DO FORM CustSearch TO ThisForm.cCust_ID
```

This is okay, as long as the desired result is a single value. However, if you need a series of customer ID's that can be navigated through using VCR-style buttons, it would be possible to do this:

```
DO FORM CustSearch WITH ThisForm
```

And the CustSearch form can store a reference to the customer form to one of its properties.

Now, the user can use the CustSearch form to filter the list of all customers to only those in Cleveland, and select the customer that showed the largest volume of purchases for last year.

When the user is finished, the CustSearch form can manipulate several properties of the Customer form. In this way, the information that the CustSearch can "return" is not limited to just the cCust_ID of the best customer in Cleveland. The CustSearch form can set a property of the calling form that indicates the number of customers in the subset the user selected (those in Cleveland), set another property of the calling form to indicate the Customer ID of a single customer in the subset, and populate an array property of the calling form with the customer ID's of all the customers in the subset. The calling form can then use this information in navigation methods and related objects to allow the user to "browse" or examine in sequence the customers in the subset.

How about the situation where I'm actually launching Form "B" from Form "A," and I want to pass some god-awful complex set of information to Form "B"? Yes, I could pass a whole bunch of parameters, or even some kind of specialized object. On the other hand, my launching code could look something like this: (Note that this contradicts some of what I've said previously, but read on.)

```
DO FORM <FormName> NAME ThisForm.oFormLaunched
ThisForm.oFormLaunched.cCust_ID = ThisForm.cCust_ID
DIMENSION ThisForm.oFormLaunched.aCustList[ThisForm.nCustomers]
ACOPY(ThisForm.aCustList, ThisForm.oFormLaunched.aCustList)
```

As I'm sure you noticed, this code represents a couple of pretty tightly coupled forms. Manipulating a form's properties requires a lot of knowledge about that form. However, this is the kind of thing that could be built into the foundation classes or application framework, so that the properties being manipulated, or the methods being called, are not unique to a single form, but part of every form of a particular type in the system.

Our technical editor expressed interest in an example of building inter-form communications techniques into a framework. Rather than bore you with a bunch of code, I thought I'd explain one approach I've used for an application.

The situation was the need (as in the text) to allow the user to select a subset of patients, or patient families, or services, or any other large set of records in a health-care facility application, and then navigate through the resulting list using VCR-type navigation buttons. Because this application would be installed as a client/server application at some sites, all data manipulation had to be done using views. In a C/S app, pulling up a list of all 10,000 patients in a picklist is something to be discouraged; however, navigation through a reasonable subset was part of the application's spec.

All data-aware forms in the framework have two properties to support this functionality: oNavList and a property to hold the key value for the current record, uKeyValue. The framework also includes a composite object, cntNavButtons, which includes the four VCR buttons, a label that indicates the number of items in the subset, and an array property to hold the ID's of the items in the subset. The cntNavButtons object also has the four navigation methods GoNext(), GoPrev(), GoFirst() and GoLast(), and a couple of numeric properties, nCurrentInList, to act *as a "pointer" to the currently selected item, and nLastInList to indicate the total number of items in the list. The navigation methods manipulate the nCurrentInList property, enable and disable the VCR buttons depending on the position within the list, transfer the record ID from the array to the form's uKeyValue property, and call the form's Requery() method to requery the current views.*

The Init() method of cntNavButtons "registers" the VCR-button object with the form by storing a self-reference to the form's oNavList property. When the user wants to launch the (modal) search form, the foundation data-aware form class's Search() method executes the search form:

```
DO FORM PatSearch WITH This TO ThisForm.uKeyValue
```

The search can simply return a single value through 'its Unload() method. It checks the passed calling form to see if it has a non-null object oNavList property. If not, it simply passes the user's selection back to the calling form's Search() method via the search form's Unload() method. If the calling form does have a non-null object oNavList property, the search form also stores the ID's of the subset to the array property of the calling form's oNavList object, sets the oNavList object's nLastInList and nCurrentInList properties, and sets returning the ID of the currently selected item in the list via the Unload() method.

As pointed out in the text, these forms are very tightly linked. However, this is tolerable because the knowledge that each form requires of the other is not unique to each form instance but to the form classes, which are common throughout the application. Thus, while the forms are tightly coupled, they are loosely coupled within the context of the application framework.

Other issues with private data sessions

One somewhat bothersome thing about private data sessions is that a bunch of environmental settings are scoped to the data session. This means that any settings you have established globally, such as SET TALK, SET DELETED, SET NEAR, and so on, immediately revert to their default values as soon as a form with a private data session is instantiated. As a result, the desired settings must be re-established for each form instance. (See the VFP help file entry for SET DATASESSION for a complete list of all data session-scoped settings.)

The brute-force approach is to simply SET TALK OFF, SET DELETED ON, and so forth, in the Init() or the Load() of each form. This requires you to remember to include all settings that are needed for each form, and place the needed code in each form. A better, and more object-oriented way, is to establish all preferred settings in a form foundation class (say, in the LOAD() event method, then simply issue a DODEFAULT() in each form and override any of these new "default" settings as needed.

A very popular approach (which I use) is to create an object class that handles this chore, and to drop an instance of this object onto the data-aware form foundation class. This approach relies on the automatic execution of the Init() and Destroy() event methods. The Init() event method calls a Set() method, which stores the current values, and by reading its own object properties, sets each environmental setting to the desired value. The Destroy() code can optionally call a Reset() method to restore all settings to their original values. This is useful when the object is used to set environmental settings in the default data session. The object classes I use for manipulating environmental settings are slightly adapted from the ones presented in the *Visual FoxPro 3.0 Codebook* by Y. Alan Griver. Just to give you a taste of how these work, **Listing 9-2** shows a few lines of code that the cSessionEnvironment object executes in its Set() method.

***Listing 9-2**. A piece of the cSessionEnvironment.Set() method.*

```
IF EMPTY(this.cCentury)
   SET CENTURY OFF
ELSE
   luTemp = this.cCentury
   SET CENTURY &luTemp
ENDIF

IF This.nCenturyTo = 0
   SET CENTURY TO
ELSE
   IF This.nCenturyToRollover = 0
      luTemp = LTRIM(STR(This.nCenturyTo))
      SET CENTURY TO &luTemp
   ELSE
      luTemp1 = LTRIM(STR(This.nCenturyTo))
      luTemp2 = LTRIM(STR(This.nCenturyToRollover))
      SET CENTURY TO &luTemp1 ROLLOVER &luTemp2
   ENDIF
ENDIF
```

If you create applications for many different clients, you might want to subclass this type of object class, setting your preferences in the subclass. On the other hand, if you are a corporate developer, you might prefer to include your preferred settings in the foundation class. Either way, overriding these settings for a particular form is no problem. Since the object in the form is an instance of the cSessionEnvironment class, you can simply change the settings by adjusting the properties in the object instance on the form that needs something set differently.

Because this object is always the first object on the form (since it's added in the form foundation class), its Init() and Set() methods always execute before any other control is instantiated, so the new settings are in effect when the rest of the controls show up for work. However, realize that a *lot* of stuff happens before the environment-setting object instantiates, some of which is affected by the settings that are manipulated via the environment-setting object.

Here is the sequence of events that occur when a form that uses the DataEnvironment is launched:

1. DataEnvironment.OpenTables()
2. DataEnvironment.BeforeOpenTables()
3. Form.Load()
4. Cursor.Init()
5. DataEnvironmnent.Init()
6. FirstObject.Init()
7. Form.Activate()

8. Form.Paint()

Note that at the time the cSessionEnvironment object instantiates, all objects associated with the data tables and their setup are up and running. The cSessionEnvironment is instantiated at event 6 in the list above. To show the effects this can have, consider two settings: TALK and DELETED. TALK defaults to "ON" and DELETED defaults to "OFF". If you have a view that is being opened in the data environment, these default values are in effect. This means that the query will not ignore deleted records, and the message "Selected x records in .028 seconds" will appear either on the status bar if STATUS BAR is set ON, or on the VFP desktop if STATUS BAR is set OFF.

In the case of the views and deleted records, the solution is to make sure you always set the NoDataOnLoad property to .T. and REQUERY() the view from the form's Init(), or, alternatively, set AutoOpenTables to .F. and call the OpenTables() method from the Form.Init().

The problem with the TALK setting is not as simple. Actually, with the status bar set ON, it might not be an issue (the user might not even notice the message), but if your application doesn't use the status bar and it's turned off, then the only work-around is to issue a SET TALK OFF in the BeforeOpenTables() method. Unfortunately, this must be done in each form. It still isn't possible to subclass the data environment where you could employ the SET TALK OFF command and have your forms use this user-defined DataEnvironment class rather than the standard VFP DataEnvironment class. You might be tempted to instead add the environment setting object via the form's Load() event method, which works fine and does indeed get the environmental settings in effect a little sooner, but not soon enough for these two issues—the tables are already open when the Load() fires.

To form or not to form—running forms as object instances

There is a school of thought that says forms should be instantiated from a class, using CREATEOBJECT() rather than using DO FORM. The arguments in favor of this line of thought make some sense. For those of us who tend to be a bit purist in our thinking when it comes to object-oriented programming, it just seems to be the "right" thing to do. Everything should be stored as an object class and instantiated at runtime. It encourages us to design our forms in a manner that allows more opportunities to subclass the forms and to specialize their functions.

This last argument becomes greatly compelling when you've just completed the last of three or four complex forms that have almost identical features and functionality. Running forms as forms rather than as object instances certainly doesn't preclude creating a form class that provides certain common features that are then specialized either in a subclass or in the form instance. However, if you get into the habit of basing all forms directly on foundation classes rather than specialized form classes, you can miss opportunities to really take advantage of the power of object-oriented programming.

Despite the "purity" and "rightness" of doing so, many believe that the disadvantages of designing forms as classes rather than form instances outweigh the advantages.

The greatest disadvantage of running forms as object instances is the fact that form classes do not have a DataEnvironment object. Because you still can't work with the DataEnvironment object class the way you can most other baseclass objects, and can't add a DataEnvironment object to form classes in the Form or Class Designers, it becomes necessary to programmatically add a DataEnvironment object, or to programmatically open all necessary tables and supporting files at runtime. While this isn't really that big a deal, it does mean that you give up the ability to create drag-and-drop form controls from the form's data environment, and the property sheet picklists for each control's ControlSource property are empty.

Some would argue that the necessity of programmatically adding a DataEnvironment object or its equivalent is a strength of running forms as class instances rather than as forms, because it rewards developing a subclassed version of the DataEnvironment that can provide far greater functionality than the native DataEnvironment object. However, this is not precluded by running forms, as you can see from the ideas presented in Chapter 7.

In the final analysis, making a decision to run forms or objects instantiated from form classes is influenced by a number of things. There is no "right" or "wrong" answer to this question, but here are some factors that are likely to influence your decision:

- The experience and skill level of the programmers working on the project. Forms are a little easier to learn to use because they include the data environment, and they're a little easier to run, test and debug. Instantiating forms from form classes is a bit more complex, often aided by builders for creating data environments or adding data manipulation classes, and may have a "deeper" class hierarchy, making it more difficult for the novice VFP developer to understand and adopt.
- The need to access more than one type of data source. Instantiating forms from object classes is sometimes done as a consequence of the need for more flexibility in the type of data environment and data manipulation classes that must be defined at runtime. Some developers, once they've decided to chuck the Visual FoxPro DataEnvironment, decide that there is little reason to recommend forms over form classes (although Chapter 7 shows that you can have your cake and eat it, too!).
- The scope of the foundation classes. Foundation classes can be developed for use in a single application or a uniform corporate development environment, or might be intended to eventually form the basis of a more robust and broadly applied application framework. Again the considerations involve ease-of-use, flexibility and adaptability, as well as the difficulty a new user might experience in learning to use the foundation classes.
- The personal preferences and inclinations of the developers working on the project. Some of us think instantiating from form classes just "feels right" and enjoy the challenge of pushing the envelope, while others prefer to use the tools as Microsoft has provided them.

Forms and delegation

As I've learned to create applications in Visual FoxPro, I've gradually learned the importance of delegating functionality to the appropriate object. This usually means delegating responsibility to form methods rather than performing complex tasks in control methods.

You should see many "red flags" when a certain pattern appears in your programming. Here are some that are commonly cited:

- A segment of code indented five to seven levels deep, indicating numerous nested IF…ENDIF and DO CASE…ENDCASE control structures—an indication that the code's logic is overly complex, and could possibly be simplified by moving some of the logic into smaller, more concise methods.

- Overriding a class method without issuing a DoDefault()—an indication that perhaps some functionality in the superclass really belongs in a subclass.
- A long list of 10 or 12 arguments to a method or function—an indication of a tightly coupled routine.
- A routine (function, procedure or method) structured as a single DO CASE, each CASE testing the value of a single parameter and then executing a large block of code depending on the value of that parameter—evidence of a routine that is possibly logically cohesive but should be broken into several appropriately named methods or routines, one for each CASE condition.

Similarly, in an object-oriented language like Visual FoxPro (and other object-based tools like Access and Visual Basic), a red flag should go up any time you see more than three to seven lines of code in a control method. I'm not referring here to method code in a control class. The method code in a reusable control class can often be quite complex. However, the programmer should recognize the limited role of a form control.

A form control performs two functions. The first is to present a single piece of information, or a set of related pieces of information. The second is to act as a locus of interaction between the user and the form.

Consider a knob on a radio. You can tell from the position of the red line engraved on the knob the approximate frequency to which the radio is tuned. You can also grasp the knob, and by turning it, change the station you're listening to. Does the knob actually change the frequency that the circuitry is tuned to? No, it's just a beige plastic thingy that is attached to a metal shaft that is in turn attached to a variable capacitor that actually does the tuning. The knob gives us a discrete piece of information and allows us to indicate our wishes. The knob is not responsible for carrying out our wishes, only for conveying our desire to the component in the radio that is designed to act on our desire to listen to the news instead of some R & B. The garage door opener on my sun visor is not responsible for actually opening the garage door. It's a control that is only responsible for conveying my desire to open or close the door to the device designed to receive that request. Neither is the receiver responsible for moving the garage door. It delegates that responsibility to the opener itself, simply sending a signal to the opener to say "HEY! Open the Door!"

Similarly, in our forms, the Save button shouldn't have 50 lines of code in 'its Click() event method that loops through all of the work areas and executes a TableUpdate() on each one. It should, instead, simply have a single line that says:

```
ThisForm.Save()
```

As another example, a text box that is used to enter an order number should not have a bunch of code in 'its LostFocus() method for establishing a view parameter and executing a REQUERY(), or selecting a work area and performing a SEEK(). The LostFocus() method might contain some check to see if the user has keyed in a new order number, and if so, it calls a Requery() or LookUp() method belonging to the form.

There are several reasons that the amount of control method code should be kept to a minimum. First, while initially it may seem that a certain block of code should only be executed in response to this *one* event, it quite often turns out that there are several events that need to trigger execution of this block of code. There's no law that says these other events can't call the method of the control, but remember that method names should bear some resemblance to the functions they perform. If the Save function is being performed by a Click() event method, and a LookUp function is performed by a LostFocus() event method, it can be a little difficult to find the code that performs a particular task.

This leads to another argument in favor of delegating important operations to form methods. When an application is in need of modification, it becomes a simple matter to open a form in the form designer, go to the methods page of the property sheet, and have in front of you all methods that do the "real work" of the form. This is especially true if you opt to display non-default properties only. You might need to do a search to determine which events are triggering which methods, but at least you have a list of all methods that accomplish the work of the form.

Another delegation issue argues against a lot of chit-chat back and forth between controls. A control might enable or disable another control based on its own value. However, if you see a long list of controls being enabled or disabled, it's likely that this function would be better delegated to a form method. If you have a control method that is enabling or disabling a whole set of controls, depending on the value of several controls, I can almost guarantee that you'll make your life much easier by moving this functionality to a form method. The form method can then be called from any event that indicates that some state has changed.

The issue of enabling and disabling controls is an opportunity to look at another mistake that I have often made. While it might be logical to have a form method that determines the "state" of the form and manipulates the appearance of specific controls in response to the current state, this kind of thing becomes difficult to maintain as controls are added or removed from a form during development. Why not make each control responsible for 'its own state? (I'll discuss this further in Chapter 11.)

Again, this is a matter of delegating responsibility to the proper object. While this chapter is about forms, not about controls, it's just as important not to overload our forms with tasks that are more easily and properly performed by the controls. The principle here is to look around and ask, "Who can I make responsible for this job?" In a way it's like being the ultimate dictatorial boss or supervisor, with perfect mindless little employees working for you. You can delegate responsibility to anyone you designate, knowing that they can't file a grievance or complain to *your* boss. They'll do whatever job you decide it's best that they do. If you decide that one employee (object) is doing too much, and needs too much of your attention to make sure that they do their job properly as conditions change, you can distribute their responsibilities among other objects that can better adapt to changing conditions, and keep the "factory" turning out widgets.

Forms as business objects

If you ever do any reading on the subject of object-oriented analysis and design, you'll be confronted with a great deal of discussion about *business objects*.

In an object-oriented software system, real-world things are modeled in the software. The things being modeled are the objects that are part of the business and what it does. These things can be employees, orders, products, customers, invoices, credit memos, purchase orders, pick tickets, time cards, patients, physicians, vendors, production lines, and on and on. In many object-oriented systems, an attempt is made to define all of these various objects and the ways in which they interact in the context of the business or enterprise. In order for the software system to function properly, it must accurately model the enterprise it is used to manage.

Each object in the system must have behaviors that accurately mimic those real-world objects they model. The various objects must interact with each other in the same manner as those in the enterprise being modeled. In modeling real-world things as business objects, the desire is to encapsulate all behaviors of the real-world thing in the software object. Keep in mind that there is often a tangible, visible aspect to the real-world objects, just as there can be for the software objects that model them. However, there are other intangible and invisible aspects to both the real-world and software objects.

Let's consider, as an example, a specific company's invoice. The tangible, visible aspect of an invoice is a piece of paper (a *document*) on which is listed all information pertinent to the invoice: the customer, the terms, the date, the purchase order number, the total, and an itemized list of the items the invoice covers.

However, an invoice could be only one aspect of another, more abstract concept. For a business in which there is a one-to-one relationship between orders and invoices, there are (at least) *two* documents, or real-world objects, that represent a different aspect of what could be viewed as the same thing, that could be referred to perhaps as a *transaction*. The order form represents a transaction from the time it is received by the company to the time that the goods have been shipped or the services delivered. The invoice represents the same abstract idea, but reflects the information needed by the company to inform its customer of the amount due for the goods or services, and the information the customer needs to process the invoice for payment. There might be other documents that reflect a different aspect of this transaction. For instance, a job ticket or pick-list might provide information that the folks on the shop floor need to properly fill the order.

Each document mentioned thus far could have a different set of information that is of interest to the various departments in the company and to the customer. This is illustrated in Table 1.

Table 1*. A hypothetical industrial products company and various documents related to a "transaction", and who might have an interest in each.*

Document	Company Departments	Customer Departments
Order Form	Sales, Customer Service	None
Job Ticket	Production, Shipping, Quality Control	Engineering
Shipper	Shipping	Receiving, Production
Invoice	Accounts Receivable, Sales	Accounts Payable

As shown in Table 1, an order form contains very little information of interest to the customer, but is of significant interest to sales and customer service. You could add an additional document called an "acknowledgement" that would be of interest to the customer's purchasing department. Similarly, an invoice is of no interest to the production department, but life revolves around the invoice for the accounting departments of both the company and its customers.

So what you have are a bunch of documents that you might be tempted to implement as separate forms in your application. However, they are really only different views of the same thing—the transaction from buyer to seller. If you can create an object that encapsulates all behaviors of this transaction-thing, you've done much of the work required by the five (or more) documents listed above. Such an object, when modeled in software, is referred to as a *business object*.

It is possible that a particular business needs only one user interface for this type of business object—one that allows order entry and displays the status of the order, from receipt to fulfillment and shipping, to invoicing, to payment.

There has been considerable discussion of creating reusable business objects with Visual FoxPro. Some frameworks have even incorporated the concept of the business object into their design. However, in most object-oriented systems, the business object is a non-visual entity that abstracts the behaviors and relationships but delegates user interaction with the object to another component in the system. The reasoning is that (to continue with the example above) a "transaction-thing" object does not have, as part of its properties or behavior, a text box to display the name of the customer. The customer placing the order is, indeed, a property of the object, but it is incorrect to assume that all instances of the transaction object will require user interaction, and therefore require a text box to display this information on-screen.

To implement this type of business object in Visual FoxPro, you would create an object class based on the Custom baseclass, and add properties and methods as appropriate to model the behavior of a transaction for this company. However, you'd like to drop the resulting object onto an "Order" form, or a "Job Ticket" form, or an "Invoice" form, add the necessary interface elements and be done with it. However, we're talking about database applications here, and an integral part of such an application is the data that allows us to refer to a *specific* invoice, or order, or job ticket. The elements provided by Visual FoxPro to abstract this part of a business object—DataEnvironment and Cursor objects, and the private data session—can't be contained by any object other than a form.

Ideally, you would have DataEnvironment objects that could be added to objects based on baseclasses other than forms, and you would be able to create objects that could declare a private data session, just as you can with forms. You could then create abstract, non-visual business objects as described above, and leave the interface to forms to which you add these business objects. However, you don't have those capabilities. Do you give up on the business object idea? Not on your life.

To do some really slick things with business objects in Visual FoxPro, you just need to change your perspective slightly and use some of VFP's tools.

What follows is a description of a business object I created for an application. It demonstrates how you can use a form as a business object without being constrained by the fact that a form is usually a visible application component intended to interact with a user. As in the example above, this object is a transaction object, and needed to perform the following tasks:

- Display an order-entry and credit-entry interface.
- Display an order-fulfillment interface (for the warehouse and shipping department).
- Print a shipper.
- Post the order.
- Un-post the order.

- Print an invoice.
- Update inventory.
- Update sales history.

I decided to "go with the flow" and just make this transaction object a form. I added methods to allow this transaction object to perform each task and interact with all required data. I then created two different interfaces for the form, each on a different page of a borderless, tabless pageframe. By passing an argument, I could select either of the two interfaces when the form was instantiated. In addition to these two "modes" of operation, I allowed a third mode. This third mode caused me to begin thinking about using a form as a business object in the first place.

Normally, transactions are called up using one of two methods. One is to pass the transaction number (on order number, an invoice number or a credit memo number) as an argument from another form that displays a list of open/completed/invoiced transactions. Another is to run the form and key the transaction number into a text box. In either case, the identifying transaction number is stored to a form property, and a form Requery() method is called, which requeries the necessary views so the transaction can be displayed or edited. If the Requery() method is unable to find any records matching the transaction number, it raises a dialog box to inform the user. After adding a new order or saving edits to an existing transaction, a WAIT "Saved…" WINDOW NOWAIT confirms to the user that the save was successful. As always, a failed save results in a message box informing the user of the failure and explaining the problem.

One function of this transaction object is the ability to post (or "un-post") an individual order, which is sometimes necessary. The process of posting changes the transaction from a pending transaction into an invoice or credit memo, and updates the product inventory to reflect the transaction. However, this company usually posts its orders in "batch" mode, once per week. Rather than writing a separate procedural program to perform this task, I used the Post() method that is part of the transaction form/object.

To implement this, I created a *third* mode of operation, which I call a "silent" mode. If the form is called with *no* arguments, the Init() sets the lSilentMode property of the form, and essentially does nothing further. I also added a RetrieveTxn() method, which does what is normally accomplished by entering a transaction ID into the text box, or passing it as an argument: It accepts this ID as an argument, stores it to the appropriate form property, and calls the form's Requery() method. I refer to this as a "silent" mode because the Requery() method does not raise any error dialogs if the transaction ID can't be found, but simply returns a logical value indicating success or failure back to the RetrieveTxn() method, which in turn returns this value to the process that called RetrieveTxn().

Now, here's all you need to do:

1. Compile a list of transactions that are ready for posting.
2. Instantiate the transaction form object *without making it visible.*
3. Pass each transaction ID in turn to the non-visible transaction form object.
4. If the transaction form object successfully retrieves the transaction, call the Post() method of the transaction form object.

The transaction form object is instantiated as a non-visible object using the following syntax:

```
DO FORM frmTxn NAME loTxn LINKED NOSHOW
```

The NAME…LINKED keywords cause the form to be instantiated to a specified memory variable. The NOSHOW keyword allows the form to be instantiated without being made visible.

The Post() method automatically calls the Save() method. If the Save() is successful, Post() returns .T. If the Save() is unsuccessful, the Cancel() method is called (to clear the buffers of changes made by the Post() method), and Post() returns .F.

Thus it's possible to create another form or procedure that compiles the necessary list, instantiates the transaction object, asks the transaction object to retrieve each transaction in the list and then post each one, keeping a tally of successful and unsuccessful posts. Because the Save() method of the form already wraps any changes in a TRANSACTION…END TRANSACTION/ROLLBACK, the half-dozen or so tables that are modified by the Post() method are committed in an all-or-nothing manner.

Listing 9-3 shows (in pseudo-code) a summary of this entire technique.

***Listing 9-3**. Pseudo-code illustrating the use of a form as a business object.*

```
* Init() Method
LPARAMETERS tcTxnNo, tcInterface
IF PCOUNT() = 0
   Store .T. to ThisForm.lSilentMode
ELSE
   Store tcTxnNo to ThisForm.cTxnNo
   Store tcInterface to ThisForm.cInterface
   If Requery() retrieves a record
      Check ThisForm.cInterface –
        (Activate Page1 of page frame if it's "ENTRY", ;)
         Or Page2 of the page frame if it's "FULFILL"
ENDIF

* RetrieveTxn() Method
LPARAMETERS tcTxnNo
Store tcTxnNo to ThisForm.cTxnNo
Call ThisForm.Requery()
If Requery() retrieves a record
   RETURN .T.
Else
   RETURN .F.
ENDIF

* Post() Method
Determine if the transaction is a credit and if the business
   rules permit issuance of the credit at this time
Change status of transaction from pending to posted
Calculate due-date based on terms and posting date
Update the customer's sales history to reflect the sale or credit
Update the inventory records to reflect the items being sold/returned
Save() the changes
If Save() is successful RETURN .T.
If Save() is unsuccessful call Cancel() and RETURN .F.
```

```
PROCEDURE BatchPost
DO FORM frmTxn NAME loTxn LINKED NOSHOW
SELECT cTxnNo FROM TxnHeaders WHERE complete INTO ARRAY laTxnList
LOOP through the array
   IF loTxn.RetrieveTxn(laTxnList[i]) ;
         AND LoTxn.Post()
      Record the success of the posting
   ELSE
      Record the failure of the posting
   ENDIF
```

Hopefully this discussion has shown how forms can be used as business objects, and how they can fulfill all of the requirements of business objects.

- Encapsulation of all business object behaviors into a single object.
- Separation of interface from the business object and the business logic it enforces.
- Utilization of the business object by other procedures and processes that need to rely on the business object's knowledge of business logic and rules, but have no need of any of the business object's user interfaces.

Passing parameters in form methods

You might have noticed in the previous section how data (e.g. the transaction number) is stored to a form property, and then another method is called. This called method then uses the data stored in the form property to perform its assigned task. It would be possible to pass this piece of data from one method to another as an argument, and for the called method to receive the data as a parameter. However, I've found it to be *much* better and *much* more flexible to rely on form properties to pass data from one method of the form to another.

Keep in mind that you are trying to encapsulate a set of methods and related data. If you pass data as method arguments, the data is not visible to all methods of the form—it is scoped to only the calling and called methods. The very fact that this data is of interest to more than one method is an indication that this piece of information should be stored in a form property. If two methods have an interest in this piece of data, there is a very good chance that a third method will eventually have an interest in this data. If you find yourself adding an LPARAMETERS command to a form method, you should immediately stop and ask yourself, "Is this method exposed to, or will it be called from, objects outside this form?" If the answer is "No," you should probably create a new form property to share the data between the calling method and the called method. In this context, look at passing an argument as an *optional* technique that might be suitable for some unusual circumstance, not the *default* technique that you use out of habit.

In procedural programming, we've long been used to using parameters to achieve loose coupling ("black boxing") of a function or a procedure. However, in an object-based or object-oriented language, you're "black-boxing" the *object*, not the object's *methods*. Because the methods and properties are inseparable from the object (that's part of what object *encapsulation* is), it's not that much different, conceptually, from establishing a memory variable at the beginning of a function that is referenced in several different control structures within the function.

I'm going to embarrass myself by telling you just how long this simple Truth has taken me to embrace. I'll frequently look at code I've written as recently as a few weeks ago in which I'm slavishly passing arguments from one form method to another. If you've been doing procedural programming as long as I have, it is indeed hard to break old habits and old ways of thinking. However, you will paint yourself into far fewer corners if you start employing simple practices such as this.

This chapter has taken a look at the form itself, rather than the controls that you place on the form. The next two chapters will dig down and talk about the form controls.