Data-Drive Your Applications

Doug Hennig

As VFP developers, we're used to storing application data in tables. However, another use for tables is to store information about application behavior. This month's article will look at how data-driving key pieces of your applications can make them more flexible and more maintainable.

Rick Hodder had a great article in the December 2001 FoxTalk called "Subclassing Existing Applications". In this article, he discussed a mechanism for creating variants of existing applications. One of the situations in which this is useful is when you have an application you've created for one client that another client could also use but with a few changes. Or perhaps you have a software product you sell to lots of customers but one of them needs a customized version. I've successfully used this technique in the past to create application variants for a variety of customers.

However, a different approach to the same idea is to data-drive your applications instead. An application is data-driven if the parts of it that differ between variants are read from data rather than hard-coded. There are a number of benefits to this approach, including the fact that you can minimize the need to subclass and change code. Instead, you put information into tables that are more easily edited and maintained.

Creating data-driven code is similar to creating object-oriented code in that you have to examine the tasks required for a particular operation and abstract them into a general set. Instead of subclassing to implement particular behavior, you create records in a table.

We'll look at several areas where you can data-drive your applications to make them more flexible and maintainable: menus, application settings, and dialogs. First, though, let's discuss some issues.

Issues

There are a couple of things to be aware of before you start data-driving everything in your application.

- Performance: Typically, data-driven code needs macro expansion or functions like EVALUATE(), TEXTMERGE(), and EXECSCRIPT() (the addition of these last two functions in VFP 7 makes it much easier to create data-driven applications). While very flexible, these functions run slower than hard-coded commands or assignments. Most of the data-driven stuff we'll examine is used at application startup, where performance isn't a huge issue. Be sure to optimize data-driven code that runs in a loop or in otherwise performance-sensitive parts of the application. Also, use data-driven techniques where it makes sense, not just because you can. For example, you could data-drive all your forms—at runtime, a blank form is filled with controls by spinning through a table containing the various properties of each control (class to use, Top, Left, Width, ControlSource, etc.) and using AddObject to add them to the form. This approach makes sense for forms that need to look different for different customers or where the user can customize the form, but that's a rarity, and these forms will definitely instantiate much more slowly than their hard-coded counterparts.
- Security: If your users are anything like mine, some of them epitomize the expression "a little knowledge is a dangerous thing." If you provide a table of application settings, you run the risk that someone poking around with Windows Explorer might accidentally (or otherwise) delete the table or change it in weird and unusual ways. You'll need to handle situations such as when the table is missing or damaged, contains illegal or out-of-bounds values, or has settings the user decides they don't want after all; a "restore to default" option would be a good thing for such situations. You also may consider making the table inaccessible outside your application, such as by encrypting the table or including it in the EXE if its contents aren't changed at runtime.

Data-Driven Menus

One of the last bastions of procedural code in VFP is the menu system. Although VFP comes with a menu generator so we don't have to hand-code menus, the generated code is static. That means menus can't be

reused, nor can they easily be changed programmatically at runtime (such as when conditions in the application change). VFP developers have asked for an object-oriented menu system for years to no avail. Fortunately, it's possible to create your own object-oriented menu system; see my article in the August 2001 issue of FoxTalk ("Objectify Your Menus") for an example. We'll use the ideas and source code from that article here.

While you can create subclasses of OOP menu classes or use procedural code to create a specific menu system from these classes, why not data-drive the menu? That makes it much easier to change the menu for a specific variant of the application: simply change the contents of the table containing the menu information and the application's menu changes.

Table 1 shows the structure of a menu table, which contains the definition of an application's menu.

Field Name	Type/Size	Description
RecType	C(1)	The type of menu object to create: "P" for pad or "B" for bar.
Active	L	.T. if this record should be used.
Order	I	The order in which the menu objects should be added to the menu.
Name	C(30)	The name to assign to the menu object (such as "FilePad").
Parent	C(30)	The name of the parent for this object (for example, the "FileOpen" bar may specify "FilePad" as its parent).
Class	C(30)	The class to instantiate the menu object from; leave this blank to use SFPad for pad objects and SFBar for bar objects.
Library	C(30)	The library containing the class specified in Class; leave this blank to use a class in SFMenu.vcx.
Caption	C(30)	The caption for the menu item; put expressions in text merge delimiters (for example, "Invoicing for <<oApp.cCustomerName>>").
Key	C(10)	The hot key for the menu item.
KeyText	C(10)	The text to show in the menu for the hot key.
StatusBar	M	The status bar text for the menu item; put expressions in text merge delimiters.
Command	M	The command to execute when the menu item is selected; put expressions in text merge delimiters.
Clauses	M	Additional menu clauses to use; put expressions in text merge delimiters.
PictFile	M	The name and path of an image file to use; put expressions in text merge delimiters.
PictReso	M	The name of a VFP system bar whose image should be used for this bar; put expressions in text merge delimiters.
SkipFor	M	The SKIP FOR expression for the menu item; put expressions in text merge delimiters.
Visible	M	An expression that will determine if the menu item is visible or not; leave it empty to always have the item visible.

Table 1. The structure of a menu table.

This table is used by CreateMenu.prg to create an application's menu. This PRG expects to be passed the name and path of the menu table and the name of the variable or property that will hold the object reference for the menu; see the August 2001 article for why this is required. CreateMenu instantiates an SFMenu item, then opens the specified table using the Order tag (which sorts the table so pads come first and bars second, and by the Order field). It then processes the table so active items are added to the menu (pads in the case of "P" records, bars under the specified parent pad in the case of "B" records) with the attributes specified in the table.

```
lparameters tcMenuTable, ;
  tcMenuObject
local loMenu, ;
  lnSelect, ;
  lcName, ;
  lcClass, ;
  lcLibrary, ;
  loItem, ;
  lcParent

* Create the menu object.

loMenu = newobject('SFMenu', 'SFMenu.vcx', '', ;
  tcMenuObject)
```

```
* Open the menu table and process all active records.

lnSelect = select()
select 0
use (tcMenuTable) order Order
scan for Active

* If this is a pad, add it to the menu.

  if RecType = 'P'
    lcName    = trim(Name)
    lcClass   = iif(empty(Class), 'SFPad', trim(Class))
    lcLibrary = iif(empty(Library), 'SFMenu.vcx', ;
      trim(Library))
    loItem    = loMenu.AddPad(lcClass, lcLibrary, lcName)

* If this is a bar, add it to the specified pad.

  else
    lcName    = iif(empty(Name), sys(2015), trim(Name))
    lcParent  = trim(Parent)
    lcClass   = iif(empty(Class), 'SFBar', trim(Class))
    lcLibrary = iif(empty(Library), 'SFMenu.vcx', ;
      trim(Library))
    loItem    = loMenu.&lcParent..AddBar(lcClass, ;
      lcLibrary, lcName)
  endif RecType = 'P'

* Set the properties of the pad or bar to the values in
* the menu table. Note the use of TEXTMERGE(), which
* allows expressions to be used in any field.

  with loItem
    if not empty(Caption)
      .cCaption = textmerge(trim(Caption))
    endif not empty(Caption)
    if not empty(Key)
      .cKey = textmerge(trim(Key))
    endif not empty(Key)
    if not empty(StatusBar)
      .cStatusBarText = textmerge(StatusBar)
    endif not empty(StatusBar)
    if not empty(SkipFor)
      .cSkipFor = textmerge(SkipFor)
    endif not empty(SkipFor)
    if not empty(Visible)
      .lVisible = textmerge(Visible)
    endif not empty(Visible)
    if RecType = 'B'
      if not empty(KeyText)
        .cKeyText = textmerge(trim(KeyText))
      endif not empty(KeyText)
      if not empty(Command)
        .cOnClickCommand = textmerge(Command)
      endif not empty(Command)
      if not empty(Clauses)
        .cMenuClauses = textmerge(Clauses)
      endif not empty(Clauses)
      if not empty(PictFile)
        .cPictureFile = textmerge(PictFile)
      endif not empty(PictFile)
      if not empty(PictReso)
        .cPictureResource = textmerge(PictReso)
      endif not empty(PictReso)
    endif RecType = 'B'
  endwith
endscan for Active
use
select (lnSelect)
return loMenu
```

To see this in action, run TestMenu.prg. It calls CreateMenu, specifying that Menu1.dbf is the menu table.

Data-Driven Application Settings

With some frameworks, you create a new application by subclassing an application class and setting properties in the Property Sheet or in code. Other frameworks use a fixed application class and a configuration table of settings instead. There are advantages to each approach, but sometimes it's just easier to change the contents of a table than to subclass and worry about what needs to be changed and where.

The storage mechanism for data-driven application settings could be an INI file, the Windows Registry, a table, or a combination of these. For example, I typically store user-specific settings in the Registry but global settings in a table. However, for applications where the user needs to configure something easily, especially from outside the application, you can't beat an INI file.

For those settings stored in a table, the approach I've taken is that rather than hard-coding the list of properties to restore from the table, I use a configuration manager so not only are the property values data-driven, so is the process of restoring them. Here's an example. Say one of the properties in your application object is the name of the application (such as "Visual Cash Register"). You've always specified that by entering it into the Property Sheet, since it isn't something that would change at runtime. Then, one of the application variants you want to create needs a somewhat different name (such as "Visual Cash Register for AccountMate"). To restore that property from the configuration table, you'd need to add code to the application object to do that. That's likely only one or two lines of code, but it still needs to be written and tested.

Using a configuration manager, the contents of the configuration table drives which properties are restored. You simply add a new record to the table to start data-driving the formerly hard-coded application name property.

SFConfigMgr, in SFPERSIST.VCX, is the configuration manager I use. Its cFileName property contains the name of the configuration table to restore settings from; the default is SFConfig.dbf (we'll look at this table in a moment). SFConfigMgr collaborates with another object, SFPersistentTable, in the same VCX (this class was discussed in my January 2000 article in FoxTalk, "Persistence Without Perspiration"), which does the actual work of obtaining a value from a table and writing to the property of an object. The only public methods of this class are Restore, which restores all the settings in the configuration table for a specific object, and Reset, which resets the manager. Due to space limitations, we won't discuss this class in any detail; feel free to examine the code yourself.

Table 2 shows the structure of the configuration table, SFConfig.dbf, and Table 3 shows some sample records for the table. Notice that the value can be an expression between text merge delimiters; the configuration manager will use TEXTMERGE() on the value, so it can contain any valid VFP expression, even one calling some function or method to determine the value, such as reading it from the Registry or an INI file. Also note that the Group field can be used to distinguish which component each setting is for. You can see in Table 3 that two different objects will restore their settings from the configuration table, the application object and a logo object.

Field Name	Type/Size	Description
Key	C(25)	The key for this record.
Target	M	The property in which to store the value.
Value	M	The value for the setting.
Group	C(25)	Used to group settings by type.
Type	C(1)	The data type for the value. Since it's stored in a memo field, the configuration manager needs to convert the string value to the proper data type.

Table 2. The structure of SFConfig.dbf.

Key	Target	Value	Group	Type
Application Name	cAppName	My Sample Application	Application	C
Version	cVersion	1.0	Application	C
Data Directory	cDataDir	K:\Apps\Data\	Application	C
Menu Table	cMenuTable	Menu.dbf	Application	C
Logo Image	Picture	<<_samples + 'tastrade\bitmaps\ttradesm.bmp'>>	Logo	C

Table 3. Settings in SFConfig.dbf for a sample application.

To restore its settings, an object simply calls the Restore method of the configuration manager, passing it the group value to restore settings from and a reference to itself, so the configuration manager can update its properties. The following code, taken from the Init method of SFApplication, instantiates SFConfigMgr into the oConfigMgr property and restores all the settings in the "Application" group.

```
This.oConfigMgr = newobject('SFConfigMgr', ;
  'SFPersist.vcx')
This.oConfigMgr.Restore('Application', This)
```

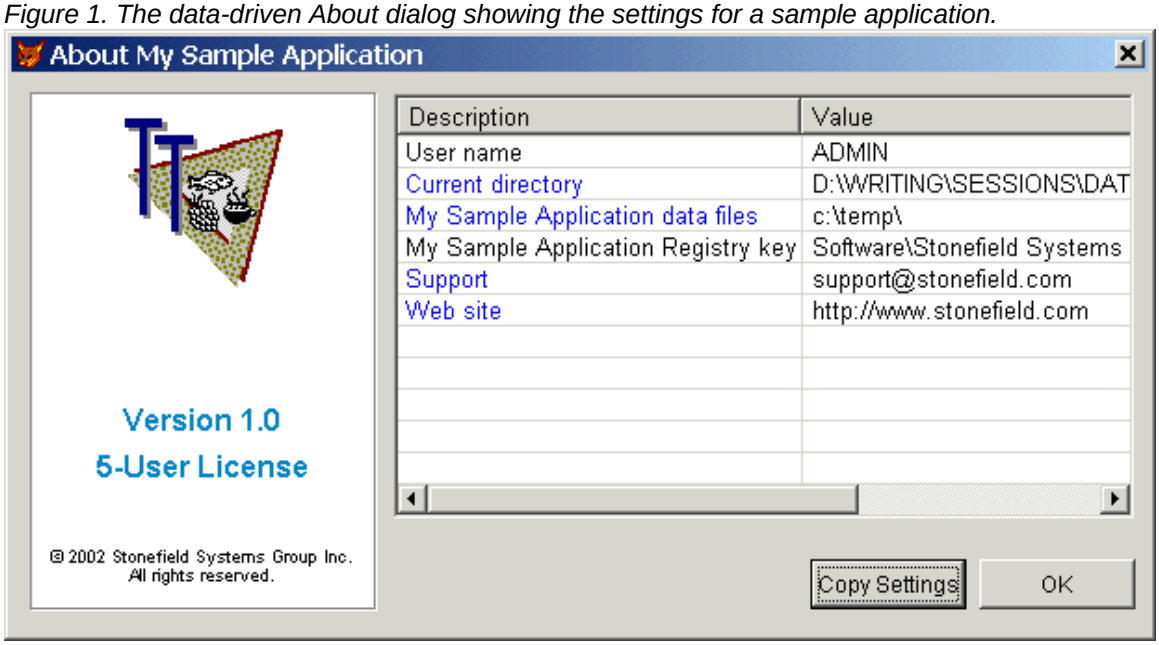
The SFLogo class uses similar code to get the name of the image file to use for the logo.

Data-Driven Dialogs

Many applications have an "about" dialog that shows things like the version number, current directory, location of the data files, name of the current user, etc. You can create an about dialog class that has the desired overall behavior and appearance and then subclass it for a specific application. However, with a data-driven about dialog, there's no need to subclass; you simply specify the things to display in the dialog with a table.

Figure 1 shows what SFAbout, a data-driven about dialog class, looks like when it displays the settings for a sample application. This class uses SFAbout.dbf to specify the settings displayed in the list and properties of the application object for other things (such as the caption, version, copyright, email address, and so forth).

Figure 1. The data-driven About dialog showing the settings for a sample application.



In addition to having a data-driven list of settings, this class has some other nice bells and whistles:

- Its size and position are saved to and restored from the Registry (using the key stored in the application object's cRegistryKey property) by the SFPersistentResizer object on the form. This object was also discussed in my January 2000 article in FoxTalk.
- The SFPersistentResizer object also handles the size and position of the form's controls when the form is resized.
- The column widths of the ListView are also saved to and restored from the Registry.

- Items in the ListView can be "hyperlinked": when the user clicks on the item, the class uses the ShellExecute Windows API function to "run" the value (I discussed this function in my July 2001 column, "Going Ape Over the Windows API"). These items are shown in blue in the list. This is most useful for things like a directory name (which launches Windows Explorer for that directory), email address (which launches the default mail client's New Message window), and Web site (which brings up that Web site in the default browser). The behavior could be changed to support calling functions within the application, such as displaying a user profile dialog when the user clicks on the user name.
- The Copy Settings button copies the settings to the Windows clipboard, which can then be pasted into an email for support.
- The logo box at the left is a class (SFLogo) that gets its information from oApp properties and from the configuration table (the name of the image file to use).

The LoadList method, called from Init, is quite simple: it opens SFAbout.dbf and spins through the active records, calling the AddSettingToList method for each one to add it to the list.

```
local lnSelect, ;
  lcValue, ;
  lcLink
lnSelect = select()
select 0
use SFAbout order Order
scan for Active
  lcValue = evaluate(ValueExpr)
  lcLink  = iif(empty(LinkExpr), lcValue, ;
    evaluate(LinkExpr))
  This.AddSettingToList(textmerge(trim(Descrip)), ;
    lcValue, Link, lcLink)
endscan for Active
use
select (lnSelect)
```

The AddSettingToList method adds the specified setting and value to the ListView control. If the item should be hyperlinked, it's displayed in blue and the target for the ShellExecute function is stored in the Tag property of the list item.

```
lparameters tcDescription, ;
  tuValue, ;
  tlHyperlink, ;
  tcLink
local lcValue, ;
  loItem
lcValue = transform(tuValue)
if vartype(tcDescription) = 'C' and ;
  not empty(tcDescription) and ;
  len(tcDescription) <= 100 and ;
  not empty(lcValue) and len(lcValue) <= 255
  loItem = This.oList.ListItems.Add(, sys(2015), ;
    tcDescription)
  loItem.SubItems(1) = lcValue
  if tlHyperlink
    loItem.Forecolor = rgb(0, 0, 255)
    loItem.Tag       = tcLink
  endif tlHyperlink
endif vartype(tcDescription) = 'C' ...
return
```

The ListViewItemClick method of the form, called when the user clicks on an item in the ListView, simply calls ShellExecute to "run" the selected item.

```
lparameters toItem
if not empty(toItem.Tag)
```

```
  declare integer ShellExecute in SHELL32.DLL ;
    integer nWinHandle, string cOperation, ;
    string cFileName, string cParameters, ;
    string cDirectory, integer nShowWindow
  ShellExecute(0, 'Open', toItem.Tag, '', '', 1)
endif not empty(toItem.Tag)
```

Summary

Although I've used data-driven techniques for more than a decade, recently I've taken it to the next level by storing much more of my application's settings in tables. These techniques have allowed me to create much more flexible applications, and create variants of these applications in a very short time and with little maintenance headaches. I hope you find these ideas thought-provoking and start thinking of ways in which you can make your applications more data-driven.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, co-author of "What's New in Visual FoxPro 7.0" and "The Hacker's Guide to Visual FoxPro 7.0", both from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals", both from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com