Session E-SCRI

Data-Driving and Scripting Applications

Doug Hennig

Stonefield Software Inc.

2323 Broad Street

Regina, SK Canada S4P 1Y9

Email: dhennig@stonefield.com

Web site: www.stonefieldquery.com

Blog: DougHennig.BlogSpot.com

Twitter: DougHennig

Overview

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 session looks at how data-driving key pieces of your applications can make them more flexible and more maintainable. In addition, we'll examine how to make your application scriptable so advanced users can customize its behavior.

Introduction

You may run into a situation where you need to create a variant of an existing application. 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, I don't use this mechanism any more. There are several reasons for this:

- You have to create a new directory and a new project file, and copy a lot of files into the new directory. You then change some of the files. This leads to duplicated files all over your hard drive.
- It can also create a maintenance nightmare, as you have to track down all the different variations of a file that you found a bug in.
- You have to create a new distribution set for the new project.
- If you have lots of customers with variants of the application, you have lots of directories and files to manage.

Instead, I prefer to use data-driven and scripted applications. An application is data-driven if the parts of it that differ between variants are read from data rather than hard-coded. It's scripted if part of the code exists in external sources that can be changed rather than inside the EXE. The benefits of using this approach are:

- You don't have to go through all the steps of creating a new project, copying files, generating a new EXE, and so on. Instead, you simply create a new set of configuration files.
- When you release a variant of the application, there's no new SETUP.EXE, just the new configuration files.
- Installation of minor changes can be easy too: there's no need to kick everyone out of the application if you just need to update the configuration files.
- There's only a small set of files to manage for each variant of the application.
- If you provide them with the capability, your customers can make changes to the configuration files to change the behavior of the application. Scripting is especially useful for this, as it allows the user to highly customize the application to meet their needs.

This document discusses some of the ways in which I data-drive and script applications today to make my applications more flexible and easier to maintain.

Data-driving applications

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.

Data-driven 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(), or EXECSCRIPT(). 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 definitely instantiate 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 is a good thing for such situations. You also may consider making the table inaccessible outside your application, such as by encrypting it.

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 use an object-oriented menu system, such as the OOP Menu project available at VFPX (http://vfpx.codeplex.com). Because of space limitations, and because it's not the focus of this document, I won't discuss how this system is implemented; feel free to examine the source code yourself.

While you can create subclasses of the 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.

Table 1. The structure of the menu table.

Field	Type	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 is 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 determines if the menu item is visible or not; leave it empty to always have the item visible.

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 holds the object reference for the menu. 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 then 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   = trim(Class)
    lcLibrary = trim(Library)
    loItem    = loMenu.AddPad(lcClass, lcLibrary, lcName)

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

  else
    lcName    = trim(Name)
    lcParent  = trim(Parent)
    lcClass   = trim(Class)
    lcLibrary = 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, such as Visual FoxExpress, you create a new application by subclassing an application class and setting properties in the Property Sheet or in code. Other frameworks, such as Visual MaxFrame Professional, 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, an XML 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 INI or XML files.

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 two other objects, a persistence object (SFPersistentTable, in the same VCX), which does the actual work of obtaining a value from a table and writing to the property of an object, and a scripting object, which allows values to be determined by executing script code for maximum runtime flexibility (we'll discuss scripting later). 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 uses 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 restore their settings from the configuration table, the application object and a logo object.

Table 2. The structure of SFConfig.dbf.

Field	Type	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 3. Settings in SFConfig.dbf for a sample application.

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

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).

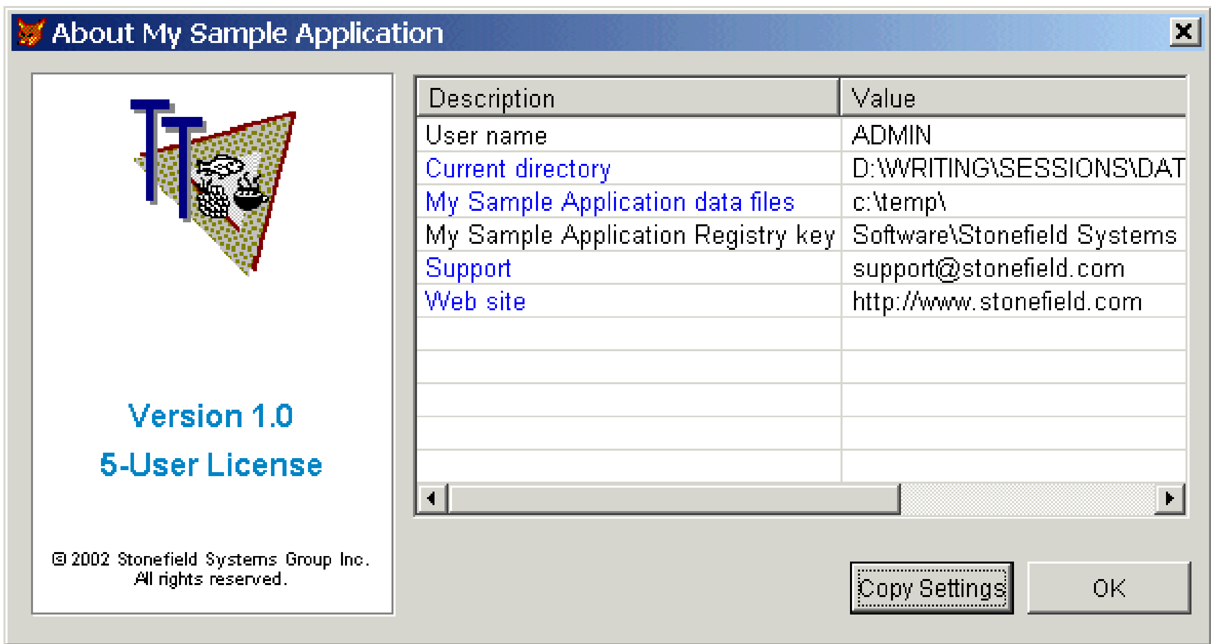

Figure 1. The data-driven About dialog showing the settings for a sample application.

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 SFPersistentForm object on the form.
- 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. 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)
```

Other data-driven ideas

There are lots of other things you can data-drive in your applications. One obvious target is a processing function which varies from implementation to implementation. The Template Method design pattern is one in which rather than putting all the code to perform a series of steps into one method, you use calls to other methods that perform the steps. That way, the method is really just a template for how the process should be done. Here's an example of a method called DoMyProcess:

```
This.OpenFiles()

This.ReadFiles()

This.ProcessFiles()

This.WriteOutput()

This.CloseFiles()
```

The benefits of this design pattern are that the code is simple, easy to read and maintain, and you can easily change the behavior of one or more steps in a subclass.

However, what if the list or order of the steps needs to change? For example, in one situation, you may need to do an extra step after the call to ProcessFiles. To do that in a subclass, you'd have to copy the code from DoMyProcess and paste it into a subclass, then add the additional step. Anytime you need to copy and paste code, alarm bells should go off in your brain.

Instead, data-drive the steps themselves. A table would contain the steps to execute and the order in which to execute them. You can easily add, change, or remove records, or change their order, to change how the process is performed.

Scripting applications

A scripted application is one that supports calling code from various places that's not built into the EXE but in external sources. For example, you can script Office applications by writing Visual Basic for Applications (VBA) code that is stored in documents. This code can be executed on demand or automatically (such as when the document is opened).

One VFP application that's heavily scripted is the Class Browser; these scripts are referred to as "add-ins". Almost every method in the browser form (Browser.scx) has code similar to the following:

```
* some code here

IF this.AddInMethod(PROGRAM())

  RETURN

ENDIF

* some code here
```

(If you want to see the source code for the Class Browser, unzip XSource.zip in the Tools\XSource subdirectory of the VFP home directory, then look in VFPSource\Browser.)

We won't look at the code for the AddInMethod method (check that out yourself if you're interested), but basically it looks at a list of registered add-ins to see if there's one for the specified method and executes the script code if so. This makes the Class Browser very extendible; VFP gurus like Steven Black and Tamar Granor have created add-ins for the Class Browser to provide features Microsoft never thought of.

A more "real world" example where scripting is useful is the case of calculating taxes. Some jurisdictions have unusual (OK, weird <g>) tax laws. For example, in Canada, the Goods and Services Tax (GST) applies to doughnuts if you buy less than six but not if you buy more than six. The rationale is that six or more qualifies it as groceries (which aren't taxed) while less than six qualifies as fast food (which is taxed). To make it even more complicated, in some provinces, the Provincial Sales Tax (PST) is calculated on the purchase price with the GST included (so you're paying tax on tax), while in most, it's calculated with the GST excluded, and one province has no PST at all. If you sell an application that deals with taxes, imagine what the DO CASE statement would look like to handle each of the possibilities in every jurisdiction your application is used! And, of course, you have to keep up with tax law changes in every jurisdiction and release an upgrade to your application when they do.

With scripting, you put the responsibility of applying and keeping up with tax laws in the hands of the users (or possibly resellers) of your application. Here's a simplistic example taken from TestTaxes.prg. If a script exists to calculate taxes, the current order is converted to XML and passed to the script, which calculates and returns the tax. If no script exists, a default calculation using a tax rate of 5% is applied to every line item in the order.

```
if loScript.DoesScriptExist('CalculateTax')
  cursortoxml('', 'lcXML', 1, 0, 0, '1')
  lnTax = loScript.Execute('CalculateTax', lcXML)
else
  calculate sum(0.05 * Unit_Price * Quantity) to lnTax
endif loScript.DoesScriptExist('CalculateTax')
```

Here's the code for the CalculateTax script in SFScript.dbf. This is for a fictional jurisdiction that applies different tax rates to different types of products, and in the case of one product type, the rate depends on the amount sold (similar to the rules for Canadian doughnuts).

```
lparameters tcXML
local lnTax, ;
  lnLineTotal, ;
  lnRate
xmltocursor(tcXML, 'Order')
lnTax = 0
scan
  lnLineTotal = Quantity * Unit_Price
  do case
```

```
    case Category_Name = 'Dairy Products'
      lnRate = 0.03
    case Category_Name = 'Seafood' and Quantity > 500
      lnRate = 0.05
    case Category_Name = 'Grains/Cereals'
      lnRate = 0
    otherwise
      lnRate = 0.045
  endcase
  lnTax = lnTax + lnLineTotal * lnRate
endscan
return round(lnTax, 2)
```

Here's another example. In one of our products, exporting to a file is normally disabled in the demo version. However, one potential customer wanted to see how exporting would work for them. This application supported a startup script that normally had no code. I create some code for the script that turned on the export feature and emailed it to him so he could evaluate that feature. Without that ability, I would have had to create a custom version of the application and email this person a multi-megabyte file.

Types of scripting

There are two types of scripting you can use: file and table.

In the case of a file-based script, your application looks for a file with a specific name or in a specific directory, and if it exists, executes the code in that file. It could be a PRG (in which case you'll need to compile it first), an FXP, or a text file that you'll use FILETOSTR() to read the contents and then EXECSCRIPT() to execute. There are a few advantages of file-based scripts:

- There's no "registration" process required: you simply create or drop the file into the appropriate directory.
- It's easily edited using Notepad, so you can talk even non-developers through any changes over the phone.

Table-based scripts exist in a table. Typically, the table has a name field so you can find the desired record and a memo field to contain the code. When a script should be executed, you find the appropriate record in the table and use EXECSCRIPT() to execute the code or write the memo out to a temporary PRG, compile the PRG to an FXP, call the FXP, and then delete the PRG and FXP files. For improved performance, you could even pre-compile the source code and store the object code in another memo, then write that memo out to an FXP and execute it without having to compile it first. Here are the advantages of table-based scripts:

- The code is relatively secure: someone needs an application that can read from and write to DBF files to view or change the script. If security is a concern, you could place the table in a

directory that all users have read access to but only certain users have write access, encrypt the file or its content, or even use SQL Server as the script repository.

- You can provide a script editor utility, which would be easier for the user to work with and in which you have additional control over the code the user enters (you can check for certain types of errors, you can add additional code such as parameters statements behind the scenes, etc.).

I typically use table-based scripts for most scripting in my applications and file-based scripts for temporary scripting. An example of the latter is diagnostics. In some situations, it may be difficult to track down the cause of a customer's problem: their data may have some bad records, their environment may be completely different from yours, the application may not have been correctly installed or configured, etc. In parts of your code particularly sensitive to these issues, you can check to see if a script file exists, and if so, execute it. The script file might do some diagnostics, dump the results to a text file or temporary table, and email the results to you. When you've resolved the problem, you simply delete the script file to turn off diagnostics.

Scripting issues

While providing great flexibility, scripting your applications comes at a price. Some of the issues related to scripting are similar to those of data-driving.

- Performance: As you likely know, calling a subroutine is slower than in-line code. It gets worse with script code because there are several steps required before the code is actually called. In the case of file-based scripts, you must check for the existence of a script file using FILE() and compile it if it's a PRG. In the case of table-based scripts, you have to find the script record (although using SEEK makes short work of that), then use EXECSCRIPT(). All these things take time. Not necessarily a lot of time, but you likely want to avoid calling a script in performance-sensitive code or within a loop.
- Security: A knowledgeable but ethically challenged user might put malicious code into a script. For example, they might change a script executed at the end of an order processing routine to email credit card information to a third party. To prevent this, you have to either shield things the user shouldn't have access to from the script code or specifically look for suspicious code (sort of like virus pattern scanning that anti-virus utilities use). However, even well-meaning users can cause problems: a tweak to a script to make it better can make it worse, not function at all, return bad results, etc. Only authorized users should be able to create or make changes to scripts.
- Data access: The script code can't assume it has access to open cursors. For example, if a form with a private data session calls a script manager to execute a script, unless the script manager was instantiated from the form, it'll "live" in a different data session, so the script code it calls won't see any of the data the form is working with. Another possibility: as we'll see later, you might want the script code to be VBScript, JavaScript, or .Net code, so that script code certainly couldn't touch any VFP data. One solution to this is to pass any data needed as a parameter to the script using arrays, objects with properties holding the data, XML, or ADO. Another solution is to use a robust object model in your application, such as that provided by most VFP frameworks or an n-tier design.

- Error handling: Handling errors in script code can be more complicated than handling errors in the rest of your application. Although the code is executed by calling a file or using EXECSCRIPT(), it's likely called from an object, so any errors will fire the Error method of that object rather than calling the ON ERROR handler. That means any object calling a script needs to ensure that it can handle any errors that may occur in that script, even though it has no knowledge of what that script will do. The solution to this issue is to use a TRY structure to ensure all errors are trapped locally.
- Language: Although we love VFP, the reality is that most people even remotely familiar with programming haven't even heard of it, let alone know how to code in it. Far more popular are languages such as VBScript, JavaScript, VB.Net, and C#. Fortunately, there are a couple of solutions for this issue. In the case of VBScript and JavaScript, use the Microsoft Script Control. This COM object can execute VBScript or JavaScript code, and can be instantiated and used from VFP. For VB.Net and C#, Craig Boyd created VFPDotNet, a library that allows .Net code to be executed from VFP (http://tinyurl.com/3cvb9r6). One of my developers, Chris Wolf, adapted Craig's code to create the DotNetScript library. We'll discuss these in more detail later.

Calling script files

As I mentioned earlier, file-based scripts are ideal for temporary things, like turning on diagnostics or perhaps implementing a patch for a problem prior to a full release of a new version of the application.

Here's some code, taken from an application object, which calls a script file, if it exists, during application startup. The application object has a cScriptDir property that contains the location of script files. In this case, if a file called Startup.ssf exists (the "ssf" extension stands for "Stonefield Script File"), its contents are read in and executed using EXECSCRIPT().

```
lcFile = This.cScriptDir + 'startup.ssf'
if file(lcFile)
  lcCode = filetostr(lcFile)
  if not empty(lcCode)
    try
      execscript(lcCode)
    catch
    endtry
  endif not empty(lcCode)
endif file(lcFile)
```

Earlier, I mentioned that I turned on the export feature of an application for a prospective customer. I did this by emailing him Startup.ssf, which just had one line of code to remove the Skip For condition from the Export function in the menu:

```
oApp.oMenu.FilePad.FileExportBar.cSkipFor = ''
```

This code's simplicity is yet another reason to love OOP-based menus!

Using table-based scripting

My implementation of table-based scripting has three components: a script table to contain the scripts, a script manager to manage the scripts, and script objects to actually execute the code.

The script table is pretty simple: it just consists of columns for the name of the script (NAME), the script type (SCRIPTTYPE, which contains 1 for VFP code, 2 for VBScript, 3 for JavaScript, 4 for C#, and 5 for VB.Net), the code for the script (the CODE memo), and a logical field to indicate if the script should be used or not (ACTIVE). It has a tag on UPPER(NAME) so SEEK can be used to locate the desired script. The script manager uses the active records in this table to fill a collection of script objects.

Next, let's look at the script classes. SFScript is a simple class based on SFCustom (a subclass of Custom) with just a few custom properties: cName, the name of the script, cCode, the code for the script, cID, the ID for the script, and nScriptType, the script type value. It has just one custom method, Execute, which executes the script; that method is empty in this class.

SFScriptVFP is a subclass of SFScript used for scripts with VFP code. It overrides the Execute method to execute the VFP code. It accepts up to ten parameters that should be passed to the VFP code; you can change this if you need more parameters. Note that only those parameters actually passed in are passed to the script code to prevent "must pass additional parameters" errors. In a runtime environment, EXECSCRIPT() is used to execute the script code. However, in a development environment, where you may want to debug the code, we'll take a different approach. It turns out that one of the easiest ways to crash FoxPro is to open the debugger from within code executed by EXECSCRIPT(). So, to avoid that, we'll write the code out to a temporary PRG file and then call that PRG.

```
lparameters tuParam1, ;

  tuParam2, ;

  tuParam3, ;

  tuParam4, ;

  tuParam5, ;

  tuParam6, ;

  tuParam7, ;

  tuParam8, ;

  tuParam9, ;

  tuParam10

local lcParms, ;

  lnI, ;

  lcFile, ;

  lcPath, ;
```

```
  luReturn, ;
  loException as Exception

* Build a list of parameters to pass.

lcParms = ''
for lnI = 1 to pcount()
  lcParms = lcParms + iif(empty(lcParms), '', ',') + 'tuParam' + ;
    transform(lnI)
next lnI

* Clear any previous error information.

This.lErrorOccurred = .F.
This.cErrorMessage  = ''

* If this is the development version, we'll copy the code to a PRG and call it
* as a function so we can properly debug it if necessary. Otherwise, use
* EXECSCRIPT() to execute it.

if version(2) = 2
  lcFile = forceext(addbs(sys(2023)) + sys(2015), 'PRG')
  erase (forceext(lcFile, 'FXP'))
  strtofile(This.cCode, lcFile)
  if not upper(sys(2023)) $ set('PATH')
    lcPath = sys(2023)
    set path to "&lcPath" additive
  endif not upper(sys(2023)) $ set('PATH')
  try
    luReturn = evaluate(lcFile + '(' + lcParms + ')')
  catch to loException
    This.lErrorOccurred = .T.
    This.cErrorMessage  = loException.Message
    This.oException     = loException
  endtry
  try
    erase (forceext(lcFile, 'FXP'))
```

```
    erase (lcFile)
  catch
  endtry
else
  try
    luReturn = execscript(This.cCode, &lcParms.)
  catch to loException
    This.lErrorOccurred = .T.
    This.cErrorMessage  = loException.Message
    This.oException     = loException
  endtry
endif version(2) = 2
return luReturn
```

SFScriptMSScript is also a subclass of SFScript; it's used for scripts that have VBScript and JavaScript code. The key to executing VBScript or JavaScript code is to use the Microsoft Script Control. This ActiveX control can be dropped on a VFP form or instantiated in code. Set the Language property to either "VBScript" or "JavaScript", call the AddCode method to hand it the code to execute, and then call the Run method to execute it. For documentation on the Microsoft Script Control, search the MSDN Web site (http://msdn.microsoft.com). One article I found useful was "Script Happens" by Andrew Clinick (http://tinyurl.com/3rq5fad).

SFScriptMSScript has two additional properties: oScript, which contains an object reference to the Microsoft Script Control object, and cLanguage, which contains the name of the language for the script code. The Execute method first ensures we have a Microsoft Script Control object by calling CheckScriptControl, which instantiates MSScriptControl.ScriptControl (the ProgID for the control) into the oScript property if not. Execute then sets the control's Language property to the value in its own cLanguage property, calls the Reset method to ensure the control is reset to a fresh state (the control may have been used by a previous call), and calls the AddCode method to provide the control with the code to execute. If there are any compile errors in the code (such as syntax errors), the CATCH block executes, setting lErrorOccurred to .T. and preventing the rest of the code from executing. Otherwise, Execute creates a list of parameters and calls the Run method of the script control. Note that the way you actually call Run is different than the documentation for the script control indicates; you simply pass the name of the function to execute (this code assumes that the code contains "Function Main"), followed by any parameters to pass to the function.

```
lparameters tuParam1, ;
  tuParam2, ;
  tuParam3, ;
  tuParam4, ;
  tuParam5, ;
  tuParam6, ;
  tuParam7, ;
```

```
  tuParam8, ;
  tuParam9, ;
  tuParam10
local lcCode, ;
  lnPos, ;
  lnSkip, ;
  lcParms, ;
  lnI, ;
  luReturn, ;
  loException as Exception
with This

* Ensure we have a script control object, then set things
* up and add the script code to it.

  if .CheckScriptControl()
    .lErrorOccurred   = .F.
    .cErrorMessage    = ''
    .oException       = .NULL.
    try
      .oScript.Language = .cLanguage
      .oScript.Reset()
      lcCode = .cCode
      lnPos  = at(ccCRLF, lcCode)
      lnSkip = 2
      if lnPos = 0
        lnPos  = at(ccCR, lcCode)
        lnSkip = 1
      endif lnPos = 0
      if lnPos = 0
        lnPos  = at(ccLF, lcCode)
        lnSkip = 1
      endif lnPos = 0
      if .cLanguage = 'VBScript'
        lcCode = stuff(lcCode, lnPos + lnSkip, 0, ;
          ‘on error resume next’ + ccCRLF)
      endif .cLanguage = 'VBScript'
```

```
    .oScript.AddCode(.cCode)

* If the code is OK, build a list of parameters to pass,
* then run the code.

    lcParms = ''
    for lnI = 1 to pcount()
      lcParms = lcParms + ',' + 'tuParam' + transform(lnI)
    next lnI
    luReturn = .oScript.Run('Main' &lcParms)
    .oScript.Reset()
  catch to loException
    .lErrorOccurred = .T.
    .cErrorMessage  = loException.Message
    .oException     = loException
  endtry
endif .CheckScriptControl()
endwith
return luReturn
```

SFScriptDotNet is similar to SFScriptMSScript, but it uses DotNetScript.ScriptEngine to execute code. Its Execute method is a little more complex. First, it accepts an array of all scripts using the same language. It needs that because one script may call another script, and all code has to be in the same assembly. Second, it has to build an array of parameters since we don't want a variable number of parameters passed to the .Net code that executes the script code.

```
lparameters taScripts, ;
  tuParam1, ;
  tuParam2, ;
  tuParam3, ;
  tuParam4, ;
  tuParam5, ;
  tuParam6, ;
  tuParam7, ;
  tuParam8, ;
  tuParam9, ;
  tuParam10
local lcMethod, ;
  lnParamCount, ;
```

```
  laParamArray[1, 2], ;
  lnI, ;
  lcCurrParam, ;
  luReturn
with This
  if .CheckScriptControl()
    .lErrorOccurred = .F.
    .cErrorMessage  = ''
    .oException     = .NULL.
    try
      lcMethod     = .oScript.GetValidIdentifier(.cName, .nScriptType)
      lnParamCount = pcount() - 1
      if lnParamCount > 0
        dimension laParamArray[lnParamCount, 2]
      else
        laParamArray = ''
      endif lnParamCount > 0
      comarray(This.oScript, 10)

* Put the parameters into an array.

      for lnI = 1 to lnParamCount
        lcCurrParam = alltrim('tuParam' + transform(lnI))
        laParamArray[lnI, 1] = evaluate(lcCurrParam)
        laParamArray[lnI, 2] = 'ValueType'
      next lnI

* The RunCode signature in .NET is:
* object RunCode(string[] scripts, int codeType, string methodName, object[,] parameters)

      if .cLanguage = 'VBDotNet'
        luReturn = .oScript.RunCode(@taScripts, 2, lcMethod, @laParamArray)
      else
        luReturn = .oScript.RunCode(@taScripts, 1, lcMethod, @laParamArray)
      endif .cLanguage = 'VBDotNet'
    catch to loException
      .lErrorOccurred = .T.
```

```
      .cErrorMessage  = loException.Message
      .oException     = loException
    endtry
  endif .CheckScriptControl()
endwith
return luReturn
```

To use DotNetScript.ScriptEngine, you have to use RegAsm, a utility that's part of the .Net framework, to register DotNetScript.DLL as a COM object. Be sure to specify /CODEBASE as a parameter. You have to run it as administrator on Windows Vista or later systems, so you may wish to create a BAT file with something like the following and run it as administrator:

```
C:\Windows\Microsoft.NET\Framework\v2.0.50727\regasm <path>\DotNetScript.dll" /codebase
```

The script manager class, SFScriptMgr, is based on SFCollection, a subclass of Collection. The cFilePath property contains the name and path of the script table, and cAlias contains the alias that the table was opened with. Calling the FillCollection method, which is done automatically when you set cFilePath via the Assign method for that property, fills the collection. FillCollection opens the script table and spins through all active records, instantiating an object of the appropriate class based on the script type. It stores the script code in the cCode property of the script object and, in the case of a non-VFP script, creates a wrapper PRG with the same name as the script so it can be called as if it's a built-in function. It then writes out the code to a PRG file in the user's temporary files folder and compiles it.

```
local lnSelect, ;
  llReturn, ;
  lcTempDir, ;
  lcCode, ;
  lcName, ;
  loScript, ;
  lcFile, ;
  lnTries, ;
  loException as Exception
with This
  declare Sleep in Win32API integer nMilliseconds
  lnSelect = select()
  llReturn = .OpenTable()
  if llReturn
    lcTempDir = sys(2023)
    scan for Active and not empty(Code)
      lcCode   = ''
      lcName   = trim(Name)
```

```
      loScript = .Add(lcName, ScriptType)
      loScript.cCode = Code
      do case

* Ignore scripts with a period in the name (ie. data object and event scripts).

        case '.' $ Name

* If this is a VFP script, grab the code.

        case ScriptType = 1
          lcCode = Code

* For all other script types, create a PRG with the same name as the script
* that's a wrapper for oApp.oScriptMgr.Execute. That way, non-VFP scripts can
* be called as if they're built-in functions.

        otherwise
          text to lcCode textmerge noshow pretext 2
            lparameters tuParam1, tuParam2, tuParam3, tuParam4, ;
              tuParam5, tuParam6, tuParam7, tuParam8, ;
              tuParam9, tuParam10
            local lcParms, ;
              lnI, ;
              luReturn
            lcParms = ''
            for lnI = 1 to pcount()
              lcParms = lcParms + ',' + 'tuParam' + transform(lnI)
            next lnI
            luReturn = oApp.oScriptMgr.Execute('<<lcName>>' &lcParms)
            return luReturn
          endtext
      endcase

* If we have code, write it out to the Windows temporary directory and compile
* it. That way, anything can run it by calling the script name as a function
* (this assumes the VFP path includes the Windows temporary directory).
```

```
      if not empty(lcCode)
        lcFile = lcTempDir + trim(Name) + '.prg'
        strtofile(lcCode, lcFile)
        lnTries = 1
        do while not file(lcFile) and lnTries < 5
          lnTries = lnTries + 1
          Sleep(1000)
        enddo while not file(lcFile) ...
        try
          compile (lcFile)
        catch to loException
        endtry
      endif not empty(lcCode)
    endscan for Active ...
  endif llReturn
  select (lnSelect)
endwith
return llReturn
```

The Add method, called from FillCollection, instantiates a script object of the desired type. In the case of SFScriptMSScript or SFScriptDotNet objects, it also instantiates a reference to an MSScriptControl or DotNetScript.ScriptEngine object if it hasn't already been done.

```
lparameters tcName, ;
  tnType
local lnType, ;
  llOK, ;
  loScript
with This
  lnType = iif(vartype(tnType) = 'N', tnType, 0)
  do case

* If we're using an MSScript object, instantiate the MSScriptControl if we
* haven't already done so and set the oScript property of the object. If we
* can't instantiate the control, remove the script object from our collection.

    case inlist(lnType, 2, 3)
      if vartype(.oMSScript) <> 'O'
```

```
        try
          .oMSScript = createobject('MSScriptControl.ScriptControl')
        catch
        endtry
      endif vartype(.oMSScript) <> 'O'
      llOK = vartype(.oMSScript) = 'O'

* If we're using a VB or C# script, instantiate DotNetScript.ScriptEngine if we
* haven't already done so and set the oDotNet property of the object. If we
* can't instantiate the control, remove the script object from our collection.

    case inlist(lnType, 4, 5)
      if vartype(.oDotNet) <> 'O'
        try
          .oDotNet = createobject('DotNetScript.ScriptEngine')
        catch
        endtry
      endif vartype(.oDotNet) <> 'O'
      llOK = vartype(.oDotNet) = 'O'

* We're using VFP script, so we're OK so far.

    otherwise
      llOK = .T.
  endcase

* Instantiate the appropriate control.

  do case
    case lnType = 1
      loScript = newobject('SFScriptVFP', 'SFScript.vcx')
      loScript.cName = tcName
      dodefault(loScript, tcName)
    case lnType = 2
      loScript = newobject('SFScriptMSScript', 'SFScript.vcx')
      loScript.cName      = tcName
      loScript.cLanguage = 'VBScript'
```

```
      loScript.oScript   = .oMSScript
      dodefault(loScript, tcName)
    case lnType = 3
      loScript = newobject('SFScriptMSScript', 'SFScript.vcx')
      loScript.cName      = tcName
      loScript.cLanguage = 'JavaScript'
      loScript.oScript   = .oMSScript
      dodefault(loScript, tcName)
    case lnType = 4
      loScript = newobject('SFScriptDotNet', 'SFScript.vcx')
      loScript.cName      = tcName
      loScript.cLanguage = 'CSharp'
      loScript.oScript   = .oDotNet
      dodefault(loScript, tcName)
    case lnType = 5
      loScript = newobject('SFScriptDotNet', 'SFScript.vcx')
      loScript.cName      = tcName
      loScript.cLanguage = 'VBDotNet'
      loScript.oScript   = .oDotNet
      dodefault(loScript, tcName)
    otherwise
      loScript = newobject('SFScript', 'SFScript.vcx')
      loScript.cName = tcName
      dodefault(loScript, tcName)
  endcase
  if llOK
    loScript.nScriptType = lnType
    loScript.cID         = sys(2015)
  endif llOK
  nodefault
endwith
return loScript
```

To determine if a script with a certain name exists, call the DoesScriptExist method; this method simply calls the Item method of the collection class to determine if that name exists in the collection or not. To actually execute the script, call the Execute method, passing the name of the script and up to ten parameters to pass to the script code (feel free to change this if you need more parameters). This method first checks to see if the specified script exists, and sets the cErrorMessage property if not. If so, it gets the script object for the script and builds a list of parameters to pass (only those

parameters actually passed in are passed to the script object). In the case of a .Net script, it calls GetArray to create an array of all scripts using the same language; it does that in case one script calls another script, since all the code has to be compiled into a single assembly. It then calls the Execute method of the object and sets its own lErrorOccurred and cErrorMessage properties to those of the script object so the caller can determine if there was a problem with the script, and returns the result of the script code.

```
lparameters tcName, ;
  tuParam1, ;
  tuParam2, ;
  tuParam3, ;
  tuParam4, ;
  tuParam5, ;
  tuParam6, ;
  tuParam7, ;
  tuParam8, ;
  tuParam9, ;
  tuParam10
local luReturn, ;
  loScript, ;
  lcParms, ;
  laScripts[1]
with This

* Reset the error information.

  .cErrorMessage  = ''
  .lErrorOccurred = .F.
  .oException     = .NULL.

* Ensure a valid script name was specified.

  if vartype(tcName) <> 'C' or empty(tcName)
    .cErrorMessage = 'Invalid script name specified'
    luReturn       = .F.

* Ensure the specified script exists. If so, build a list of parameters to
* pass.
```

```
  else
    loScript = .Item(tcName)
    if vartype(loScript) = 'O'
      lcParms = .CreateParameters(pcount() - 1)
      if inlist(loScript.nScriptType, 4, 5)
        .GetArray(@laScripts, loScript.nScriptType)
        if empty(lcParms)
          luReturn = loScript.Execute(@laScripts)
        else
          luReturn = loScript.Execute(@laScripts, &lcParms)
        endif empty(lcParms)
      else
        luReturn = loScript.Execute(&lcParms)
      endif inlist(loScript.nScriptType, 4, 5)
      .lErrorOccurred = loScript.lErrorOccurred
      .cErrorMessage  = loScript.cErrorMessage
      .oException     = loScript.oException
    else
      .cErrorMessage = 'Script ' + tcName + ' does not exist'
      luReturn       = .F.
    endif vartype(loScript) = 'O'
  endif vartype(tcName) <> 'C' ...
endwith
return luReturn
```

Check it Out

TestScript.prg tests the script manager by executing a VBScript script that displays a message box and changes the caption of a passed form. Here's the code from this PRG:

```
loScript = newobject('SFScriptMgr', 'SFScript')
loScript.cFilePath = 'SFScript.dbf'
loForm = createobject('form')
loForm.Show()
loForm.Caption = 'this is a test'
lcValue = loScript.Execute('TestScript', loForm)
messagebox(lcValue)
messagebox('Form caption is now ' + loForm.Caption)
```

Here's the VBScript code taken from the "TestScript" record in SFScript.dbf:

```
function Main(Form)

  msgbox "Form caption was " & Form.Caption

  Form.Caption = "Hello from VBScript"

  Main = "My return value"

end function
```

TestTaxes.prg shows an example mentioned earlier: scripting tax calculations. Here's the code that calls the CalculateTax script we looked at earlier:

```
if loScript.DoesScriptExist('CalculateTax')

  cursortoxml('', 'lcXML', 1, 0, 0, '1')

  lnTax = loScript.Execute('CalculateTax', lcXML)

else

  calculate sum(0.05 * Unit_Price * Quantity) to lnTax

endif loScript.DoesScriptExist('CalculateTax')

messagebox(lnTax, 0, 'Total Tax for Order 2')
```

Here's an example of a C# script named GetTaxRate. Note that the name of the method in the script must match the name of the script:

```
public static double GetTaxRate(string category)

{

    double rate;

    switch (category)

    {

        case "Dairy Products":

            rate = 0.03;

            break;

        case "Seafood":

            rate = 0.05;

            break;

        case "Grains/Cereals":

            rate = 0;

            break;

        default:

            rate = 0.045;

            break;

    }

    return rate;
```

```
}
```

The following code calls that script to get the tax rate for a certain category of product:

```
lnRate = loScript.Execute('GetTaxRate', 'Seafood')
```

Summary

Although I've used data-driven techniques for more than twenty years, over the past decade I've taken it to the next level by storing much more of my application's settings and code in tables. Scripting has especially been a powerful technique, and I'm using this in every application I write today. 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 flexible.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 to 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).

stonefield
Database
TOOLKIT
stonefield
Query

VFPx
MVP
Microsoft
Most Valuable
Professional
FoxRockX