# Creating a Plug-in Architecture for Your Applications

*Doug Hennig*
*Stonefield Software Inc.*
*Email: dhennig@stonefield.com*
*Corporate Web sites: www.stonefieldquery.com*
*www.stonefieldsoftware.com*
*Personal Web site : www.DougHennig.com*
*Blog: DougHennig.BlogSpot.com*
*Twitter: DougHennig*

*Adding support for plug-ins to your applications has a lot of benefits: users can extend or alter the functionality of the application, you can deploy new features without installing a new build, and you can create customer-specific versions of an application without endless sets of CASE statements. This document looks at how adding plug-in support can help your applications and looks at several techniques that can be used independently or together.*

## Introduction

Wikipedia states that "a plug-in (or add-in / addin, plugin, extension or add-on / addon) is a software component that adds a specific feature to an existing software application. When an application supports plug-ins, it enables customization." (http://tinyurl.com/7bndc7n) For example, plug-ins have proven to be extremely popular in the web browser world, adding new features such as managing passwords, anti-virus scanning, and opening files within the browser such as PDFs.

Why would you want to support plug-ins in your applications? There are several reasons:

- You don't have to think of and implement every feature every user would ever want. Instead, you can focus on the core feature set and let others come up with interesting new capabilities. If a plug-in proves to be popular, its functionality can be added to the core code (or not).
- You don't have to be an expert at everything the application might do. You can let others who are develop best-of-breed plug-ins.
- You can deploy new features without creating and installing a new build. For example, in my company's flagship product, Stonefield Query, exporting to a file is disabled in the demo version. However, one potential customer wanted to see how exporting would work for them. We created a plug-in that turned on the export feature and emailed it to him so he could evaluate that feature. Without that ability, we would have had to create a custom version of the application and email him a multi-megabyte file.
- In the introduction to her Southwest Fox 2008 session titled "Customizing Your Vertical Market Application," Cathy Pountney stated "Writing a vertical market application can be very rewarding. You write one application, sell it numerous times, and sit back while the money rolls in. Well, that's the theory anyway. The reality is that often times, new clients want to buy your software, as long as you can change this one little thing. Managing custom code for various clients within your application can easily turn into a nightmare as your client base expands." Plug-ins allow you to avoid creating separate builds for each customer and instead implement custom functionality individually. For example, we have several versions of Stonefield Query customized for different products, such as the Sage 300 accounting system and the ACT contact management system. Originally (15 years ago), these versions had customized code bases. However, that was difficult to manage: changes in one product had to be duplicated in another. So we created the Stonefield Query SDK, which was designed from the ground up to be customizable through plug-ins rather than changes to core code.

An example where plug-ins are 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, some provinces

have a separate Provincial Sales Tax (PST), some have a sales tax combined with the GST (Harmonized Sales Tax or HST), and one has no additional tax 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 plug-ins, 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 plug-in to calculate taxes exists, the current order is converted to XML and passed to the plug-in, which calculates and returns the tax. If no plug-in 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 one such CalculateTax plug-in. 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)
```

A different jurisdiction might have much simpler, or even more complex, code. The point is that each customer can have tax calculating code that works specifically for them.

## Plug-in architecture

There are lots of ways to implement plug-ins in your applications. Let's look at a few examples.

### The Class Browser

The Class Browser utility was designed to be extended with add-ins. (As Wikipedia notes, "add-in" is synonymous with "plug-in.") There are two aspects to this:

- Everything you can do in the Class Browser is handled in custom methods rather than in events like Click. This design means you can do anything programmatically without using the user interface.
- The Class Browser calls add-ins from many places: just about every method of the form and the controls in the form has code like this:

```
* Code that always executes
IF this.AddInMethod(PROGRAM())
   RETURN
ENDIF
* Code that only executes if an add-in wasn't executed
```

(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.)

The AddInMethod method looks in the browser table (by default, Browser.dbf in the VFP home folder, which on a Windows Vista or later system is actually virtualized to C:\Users*UserName*\AppData\Local\VirtualStore\Program Files (x86)\Microsoft Visual FoxPro 9\Browser.dbf) for all records with TYPE = "ADDIN," ID = "METHOD," METHOD = "*" (meaning a global add-in that executes on every call) or METHOD = the passed-in name, and FILEFILTER is either empty or contains a file specification that matches the file currently open in the Class Browser (meaning add-ins can be specific for certain files only). For each record that matches, AddInMethod calls DoAddIn to execute the add-in. Add-ins can be a class (the CLASSNAME and CLASSLIB fields contain the appropriate information), a form (the PROGRAM field contains the name of the form, including an SCX extension), or a program (the PROGRAM field contains the name of the program). All add-ins must accept at least one parameter: a reference to the Class Browser form, which allows them access to all public properties and methods of the Class Browser.

AddInMethod returns .T. if at least one add-in was called and the lNoDefault property is .T. As you can see in the code above, most calling methods only execute the rest of the code if AddInMethod returns .F. Since AddInMethod specifically sets lNoDefault to .F. before calling any add-ins, the normal behavior is that after calling an add-in, the rest of the code in the calling method executes. If an add-in wants to prevent that, it sets lNoDefault to .T. before returning.

ClassBrowserAddinResize.PRG shows how an add-in can override the default behavior:

```
lparameters toBrowser
toBrowser.lNoDefault = .F.
```

To register an add-in (meaning add a record for it to the browser table), use the AddIn method. For example, to register ClassBrowserAddinResize so it executes when the user resizes the Class Browser window, open the Class Browser and type the following in the Command window:

```
_oBrowser.AddIn('My first add-in', 'ClassBrowserAddInResize', 'Resize')
```

(The "Class Browser Methods" topic in the VFP help gives the complete syntax for the AddIn method.)

Resize the Class Browser window and notice that the controls resize as expected. Then edit ClassBrowserAddinResize.PRG and set lNoDefault to .T. Now when you resize the window, the controls don't resize.

You can also execute an add-in programmatically by calling DoAddIn and passing it the add-in name.

If you don't specify the third parameter, the add-in appears in the Add-ins shortcut menu (right-click and choose Add-ins to display that menu) with the first parameter to AddIn being the item shown in the menu.

Of course, a smart way to write an add-in is to have it self-register. **Listing 1** shows some of the code in Rick Schummer's CBChangeFont Class Browser add-in (available from http://tinyurl.com/pojnkx2) that allows you to change the font used by the Class Browser. If the program isn't called from the Class Browser, the code either uses AddIn to register itself if the Class Browser is running or writes directly to Browser.dbf if not.

**Listing 1**. An add-in can register itself automatically.

```
* Self registration if not called from the Class Browser
IF TYPE("toBrowser")= "L"
   lcName    = "Rick Schummer's Font Changer"
   lcComment = "Developed by RAS for online forum discussion and example"

   IF TYPE("_oBrowser")= "O"
      * If Class Browser is running, use Addin() method
      _oBrowser.Addin(lcName, STRTRAN(SYS(16),".FXP",".PRG"), "ACTIVATE", , , ;
          lcComment)
   ELSE
      * Use the low level access of the Browser registration table
      IF FILE(HOME() + "BROWSER.DBF")
         lcOldSelect = SELECT()

         USE (HOME() + "BROWSER") IN 0 AGAIN SHARED ALIAS curRASDateChanger
         SELECT curRASDateChanger
         LOCATE FOR Type = "ADDIN" AND Name = lcName

         IF EOF()
```

```
            APPEND BLANK
         ENDIF

         * Always replace with the latest information
         REPLACE Platform WITH "WINDOWS", ;
                 Type     WITH "ADDIN", ;
                 Id       WITH "METHOD", ;
                 Name     WITH lcName, ;
                 Method   WITH "ACTIVATE", ;
                 Program  WITH LOWER( STRTRAN( SYS(16), ".FXP", ".PRG")), ;
                 Comment  WITH lcComment
         USE

         SELECT (lcOldSelect)
      ELSE
         MESSAGEBOX("Could not find the table " + HOME() + "BROWSER.DBF" + ;
             ", please make sure it exists.", 0 + 48, _screen.Caption)
      ENDIF
   ENDIF

   RETURN
```

To un-register an add-in, call AddIn with NULL as the second parameter:

```
_oBrowser.AddIn('My Resize add-in', NULL)
```

For more details on Class Browser add-ins, see *Advanced Object Oriented Programming with Visual FoxPro 6.0* by Markus Egger, available from Hentzenwerke Publishing (www.hentzenwerke.com).

The pros of the Class Browser approach are:

- It's extremely extensible. An add-in can access all properties and methods and can override the behavior of any function.
- An add-in has full access to the Class Browser user interface so it can hide some controls, add new controls, or completely change the user interface as it sees fit.
- Add-ins are executed automatically; there's nothing the user has to do to launch one.

The cons are:

- It requires programmatic registering of add-ins. However, since it's a developer and not an end-user tool, that really isn't a problem, and if necessary, a simple user interface could easily be created.
- From a programming point-of-view, it's a lot of work to set up, since every method has to call AddInMethod.
- It's possible to completely change the user interface. While that isn't a problem for the Class Browser, an end-user application may wish to limit what things it allows a plug-in to change.

## HTML Help Builder

I've used Rick Strahl's West Wind HTML Help Builder for more than 15 years. It makes creating CHM help files easy and, dare I say it, fun (OK, not fun, but at least bearable). HTML Help Builder is extensible in numerous ways, included user-definable templates, customizable style sheets, user-defined fields, and like the Class Browser, add-ins. The main difference between Class Browser and HTML Help Builder add-ins is that the latter have to be manually invoked by the user from the Add-Ins item in the Tools menu.

To register an add-in, choose Add-In Manager from the Tools menu; see **Figure 1**. Click New to register an add-in and specify the descriptive name and comments. The choices for Add-in Type are .NET (a .NET DLL), FOX (a PRG, EXE, or APP), and COM (a COM DLL). Specify the file containing the add-in, the class within the file, and the name of the method to call. Click Save to save the add-in and add it to the Add-Ins menu.

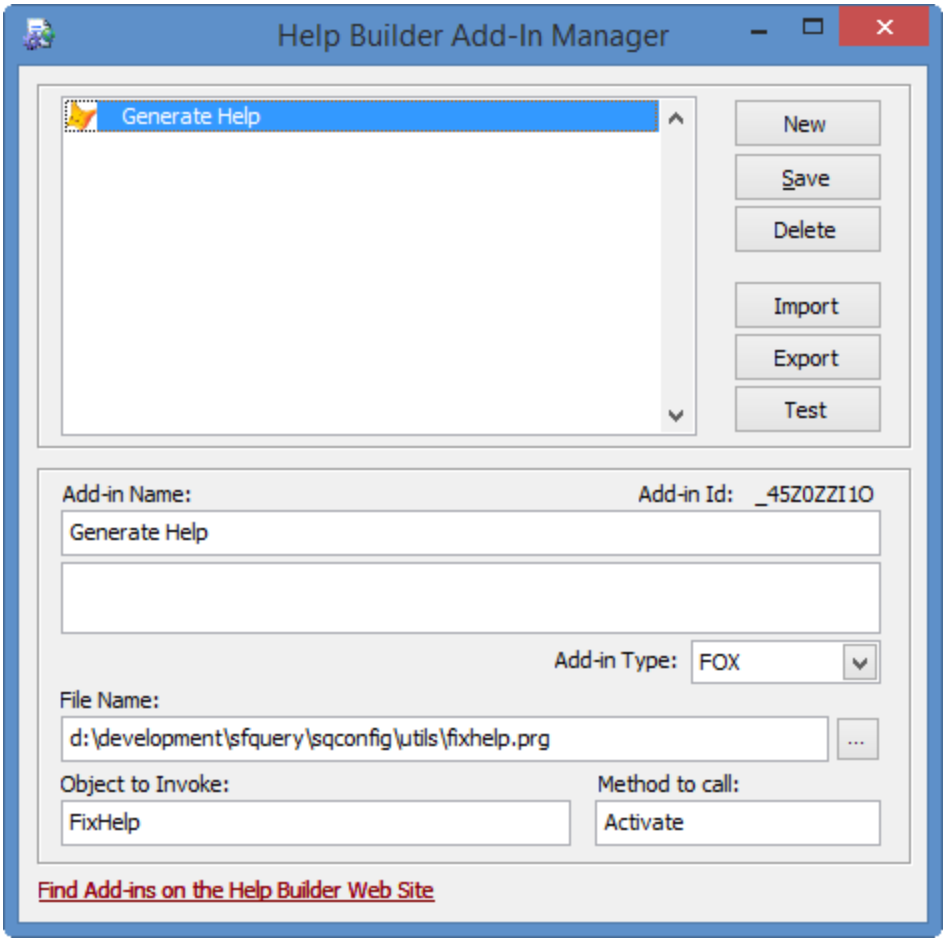



**Figure 1**. The Help Builder Add-In Manager allows you to add plug-ins to HTML Help Builder.

Like the Class Browser, HTML Help Builder stores information about registered add-ins in a table; in this case, C:\Users*UserName*\Documents\Html Help Builder Projects\Addins.dbf.

Also like the Class Browser, HTML Help Builder passes add-ins a reference to the HTML Help Builder form. This allows the add-in to modify the user interface if desired, but it's more likely an add-in will use the oHelp member of the form, which is a reference to the help builder engine that does all the actual work in HTML Help Builder.

**Listing 2** shows the code for FixHelp.PRG, an add-in I use with HTML Help Builder. This add-in does three things:

- Enables searching within the HTML files generated when the help is built (there's an option for that in the build dialog but sometimes I forget to turn it on).
- Uses the icon I want for the main topic as opposed to the one HTML Help Builder uses by default.
- Creates the CHM file and copies it up one folder level because I typically put my HTML Help Builder project files in an HTMLHelp subdirectory of the application folder.

To use this add-in, I click the Build Help button in the toolbar, select *Don't build Help File* because the add-in creates the CHM file later, and click Finish to generate the HTML files for the help project. I then choose Generate Help (the name of my add-in) from the Add-Ins menu and when it's finished, the CHM file is in the parent directory of the project folder.

**Listing 2**. FixHelp.PRG is an HTML Help Builder add-in that generates a CHM file the way I want. (Some of the code is omitted for brevity.)

```
lparameters tcPath
loFixer = createobject('FixHelp')
loFixer.Activate(tcPath)

#define CSIDL_PROGRAM_FILES 0x0026
#define HKEY_LOCAL_MACHINE  -2147483646

define class FixHelp as custom
   function Activate(toHelpForm)

* Get a reference to the help object, then figure out the path for the current
* project.

      if vartype(toHelpForm) = 'C'
         lcProjectFile = toHelpForm
      else
         loHelp        = toHelpForm.oHelp
         lcProjectFile = loHelp.cFileName
      endif vartype(toHelpForm) = 'C'
      lcPath = addbs(justpath(lcProjectFile))

* Turn on searching in case it wasn't turned on when the files were generated.

      lcFile = lcPath + 'index2.htm'
      lcText = filetostr(lcFile)
      lcText = strtran(lcText, 'var AllowSearch = false;', ;
```

```
            'var AllowSearch = true;')
         strtofile(lcText, lcFile)

* Remove the image number for the root node so it defaults to "auto".

         lcFile = forceext(lcProjectFile, 'hhc')
         lcText = filetostr(lcFile)
         lcText = strtran(lcText, '<param name="ImageNumber" value="11">' + ;
            chr(13) + chr(10), '', 1, 1)
         strtofile(lcText, lcFile)

* Find the location of HTML Help Workshop.

         lcProgramFiles = This.GetSpecialFolder(CSIDL_PROGRAM_FILES)
         lcRegVCX       = addbs(lcProgramFiles) + ;
            'Microsoft Visual FoxPro 9\FFC\Registry.vcx'
         loRegistry     = newobject('Registry', lcRegVCX)
         lcHHPath       = ''
         lcKey          = '\Microsoft\Windows\CurrentVersion\App Paths\hhw.exe'
         llGotPath      = loRegistry.GetRegKey('Path', @lcHHPath, ;
            'SOFTWARE' + lcKey, HKEY_LOCAL_MACHINE) = 0

* Compile the CHM file if we found it. Log the results.

         if llGotPath
            lcHHPath      = '"' + This.ShortPath(forcepath('hhc.exe', lcHHPath)) + '"'
            lcProjectFile = '"' + forceext(lcProjectFile, 'hhp') + '"'
            lcLogFile     = '"' + addbs(justpath(lcProjectFile)) + 'log.txt"'
            erase (lcLogFile)
            lcBatFile = '"' + lcPath + 'runme.bat"'
            strtofile(lcHHPath + ' ' + lcProjectFile + ' > ' + lcLogFile + ;
               chr(13) + chr(10) + 'pause' + chr(13) + chr(10), lcBatFile)
            declare integer ShellExecute in SHELL32.DLL ;
               integer nWinHandle, string cOperation, string cFileName, ;
               string cParameters, string cDirectory, integer nShowWindow
            ShellExecute(0, 'Open', lcBatFile, '', '', 1)

* Copy the CHM file up one folder.

            lcCHMFile = forceext(lcProjectfile, 'CHM')
            lcNewFile = forcepath(lcCHMFile, fullpath('..\', lcCHMFile))
            if file(lcCHMFile)
               copy file (lcCHMFile) to (lcNewFile)
            endif file(lcCHMFile)
         else
            messagebox('Cannot locate HTML Help Workshop')
         endif llGotPath
         return .T.
      endfunc
enddefine
```

The pros of the HTML Help Builder approach are:

- Registering add-ins is done through a simple user interface.

- Programmatically, it's easy to set up because there's just the registration table, the user interface, and the menu of registered items.
- An add-in has full access to the HTML Help Builder user interface so it can hide some controls, add new controls, or completely change the user interface as it sees fit.

The cons are:

- Add-ins have to be executed manually through a menu item.
- It's not as extensible as the Class Browser: you can't override the behavior of existing functions; you can only supplement their behavior.
- It's possible to completely change the user interface. Since HTML Help Builder is mostly a tool for advanced users, that isn't as big of a problem as it would be for an end-user application.

## Stonefield Query

My company's main product, Stonefield Query, is an end-user ad-hoc reporting solution. In order to work with any database, it needs to be extremely flexible and extensible. There are several ways plug-ins work in Stonefield Query:

- *Scripting*: you can write code that Stonefield Query executes at various places. These scripts are defined in Stonefield Query Studio, the tool used by a developer to customize Stonefield Query for an application's database (**Figure 2**). Scripts are usually written using VFP code but can also be written in VBScript, JavaScript, VB.NET, or C#. There are three types of scripts:
    - Data object scripts, which execute when determining the connection information for the database, when connecting to a database, or when retrieving data from a virtual table.
    - Event scripts, which execute at certain places in the application's lifetime, such as at startup, before the user logs in, before a report is run, and so on.
    - User scripts, which don't execute automatically but are called from other scripts or when calculated fields are evaluated.
- *Named code files*: at certain points, the application looks for code files with specific names. For example, during setup, if a file named Setup.sqs exists ("SQS" stands for "Stonefield Query Script"), the code in that file executes. Also, if a file named RepProcs.prg exists, Stonefield Query uses SET PROCEDURE TO to make the functions in that file accessible anywhere in the application that expressions can be used, such as in the header of a report. The difference between scripts and code files is that scripts are stored in a table and therefore aren't accessible to the user at runtime while code files are just text files the user can create if necessary.
- *Functions*: a special folder named Functions can contain PRG files. These PRGs are mainly used as user-defined functions for calculated fields defined by the user but can also be used anywhere else an expression can use used in Stonefield Query

because the application uses SET PATH TO *Program Folder*\Functions early in its startup.

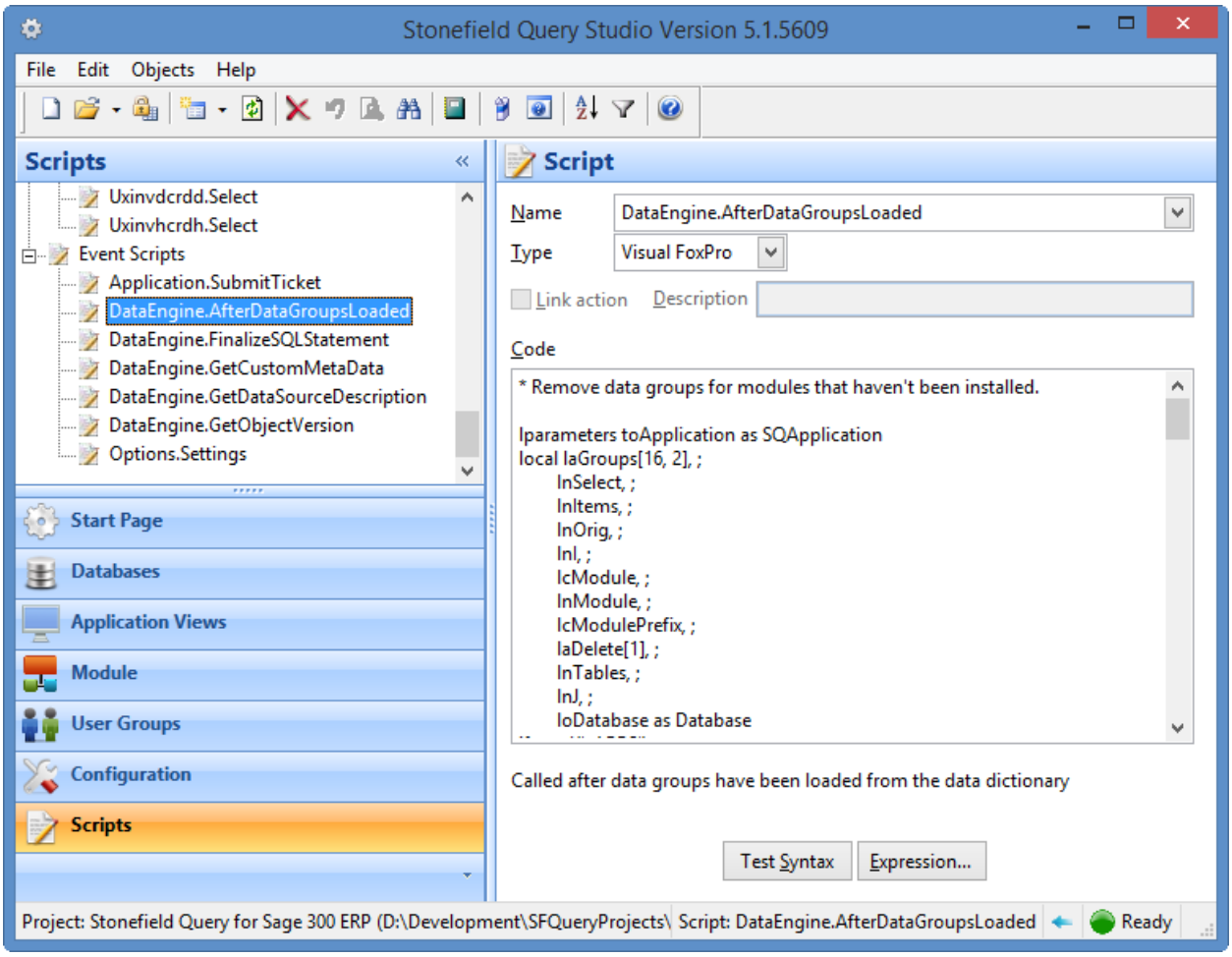



**Figure 2**. You can create scripts to customize the behavior of Stonefield Query.

The pros of the Stonefield Query approach are:

- Scripts are created in a user interface with code editing features, including IntelliSense for both VFP syntax and the application's object model.
- Plug-ins (script or code file) have full access to the Stonefield Query object model, either by being passed a reference to the application object or accessing it through a global variable.
- Plug-ins are executed automatically, either because they're called from certain events or because they're called when an expression is evaluated.
- Plug-ins have limited access to the user interface (mostly just the menu) so they can't change much (this is both an advantage and a disadvantage).

The cons of the Stonefield Query approach are:

- It's not as extensible as the Class Browser: you can override the behavior of some existing functions but not all.
- Plug-ins have limited access to the user interface so they can't change much. Of course, this could be changed if necessary by exposing the user interface objects to plug-in code.
- Because the end-user can create code files at runtime, it's possible that a malicious user could write code that damages files or does other mischief. This can be mitigated by limiting access to the physical folders where the code files are stored.

The rest of this document examines the Stonefield Query approach in more detail because I think it's more applicable to end-user applications than the others.

## My plug-in architecture

There are two types of scripting you can use: file and table. As noted earlier, Stonefield Query uses both.

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 contents, 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.

- *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 cursors. 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 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 does. 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 JavaScript, VB.NET, and C#. Fortunately, there are a couple of solutions for this issue. In the case of 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 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 Stonefield Query for a prospective customer. I did this by emailing him Setup.sqs, which had just one line of code to remove the Skip For condition from the Export function in the menu:

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

This code's simplicity is just one reason to love OOP-based menus such as those provided by the OOP Menu VFPX project (http://tinyurl.com/p2nlnmd).

## Table-based scripts

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 (shown in **Figure 3**) 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.

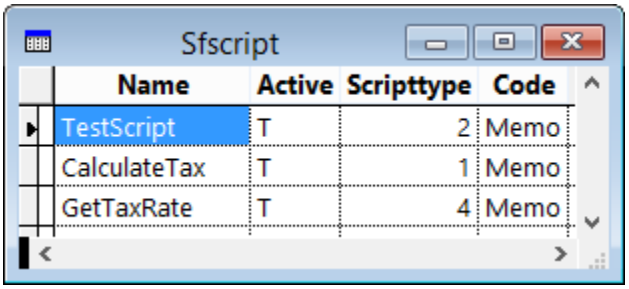



**Figure 3**. The script table has a simple structure.

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. It also has lErrorOccurred and cErrorMessage properties (which are actually defined in SFCustom) which contain information about any errors that occur during code execution.

SFScriptVFP is a subclass of SFScript used for scripts with VFP code. It overrides the Execute method (**Listing 3**) 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 VFP 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.

**Listing 3**. SFScriptVFP.Execute executes VFP script code.

```
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(juststem(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 (**Listing 4**) 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.

**Listing 4**. SFScriptMSScript.Execute executes VBScript and JavaScript code.

```
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.

         if not .lErrorOccurred
            lcParms = ''
            for lnI = 1 to pcount()
               lcParms = lcParms + ',' + 'tuParam' + transform(lnI)
            next lnI
            luReturn = .oScript.Run('Main' &lcParms)
         endif not .lErrorOccurred
         .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 (**Listing 5**) 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.

**Listing 5**. SFScriptDotNet.Execute executes VB.NET and C# 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(.oScript, 10) && zero-based arrays passed by reference

* 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
```

Of course, it's better to install and register this DLL on a customer's system using your application's installer rather than a BAT file.

The script manager class, SFScriptMgr, is based on SFCollection, a subclass of Collection. The cFilePath property contains the name and path of the scripts table, and cAlias contains the alias the table was opened with. (If you have more than one scripts table, set cFilePath to a carriage return delimited list of them.) Call the FillCollection method, which is done automatically when you set cFilePath via the Assign method for that property, to fill the collection.

FillCollection (**Listing 6**) opens the scripts table, goes through the active records, and adds an instance of a subclass of SFScript to the collection for each one. It also writes the code for the script to a PRG with the name of the script (for example, if the script is named GetCSZ, the PRG is named GetCSZ.prg) located in the user's temporary files folder, which the application includes in the path. The reason for doing that is so the script can be executed as a user-defined function anywhere an expression is evaluated in addition to being executed through the script manager. There are a couple of wrinkles with this:

- A script with a period in its name isn't written out because it's expected to be an event script, which is only executed through the script manager.
- For non-VFP scripts like C#, the generated PRG doesn't contain the code to execute, since VFP can't execute it directly, but is instead a wrapper that asks the script manager to execute the code.

**Listing 6**. SFScriptMgr.FillCollection adds script objects to the collection.

```
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)
         if vartype(loScript) = 'O'
            loScript.cCode       = Code
            loScript.nScriptType = ScriptType
            loScript.cID         = sys(2015)
            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)
         endif vartype(loScript) = 'O'
      endscan for Active ...
   endif llReturn
   select (lnSelect)
endwith
return llReturn
```

The Add method (**Listing 7**), 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.

**Listing 7**. The Add method instantiates the appropriate subclass of SFScript and adds it to the collection.

```
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.

   loScript = .NULL.
   do case
      case not llOK
      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)
   endcase
   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, shown in **Listing 8**, 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.

**Listing 8**. SFScriptMgr.Execute executes the specified script.

```
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
```

Other methods of SFScriptMgr are for script management, including SaveItem, which saves a script, and Remove, which removes a script from the collection and deletes it from the table.

## 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')
```

## Real examples

Stonefield Query for Sage Pro is a version of Stonefield Query customized for the Sage Pro accounting system. Sage Pro stores its data in either DBF files or in SQL Server; which is used is determined by a setting in Pro.ini. Sage Pro uses two or more databases: one for "system" data (data not specific to a certain company) and the one for each company the application keeps accounting information for. So, Stonefield Query needs to know two things:

- How to connect to the system database: using native access for VFP or SQL passthrough for SQL Server.
- Which database (companies) the user can report on.

It does this through a ProData.GetDataSources event script that executes at startup. The script looks at Pro.ini so it knows whether VFP or SQL Server is used (and in the case of SQL Server, what DSN to use to connect to the database), opens the table containing company information (USE for VFP data, SQLEXEC() for SQL Server), and populates a collection of "data sources" (our terminology for connection to a specific database) the user can report on.

The user can customize the Sage Pro database to some extent: they can specify the length and format for certain fields, such as GL account numbers, inventory part numbers, etc. We want to support that customization in Stonefield Query, so another event script named DataEngine.GetCustomMetaData opens the Sage Pro customization tables and goes through the records in them, updating our data dictionary as necessary.

Older versions of Sage Pro included a report writer but that was discontinued after version 6.5. However, some users may still have reports from that report writer that they need to run. We created a utility that converts Sage report writer reports to Stonefield Query reports and wrote an Application.SetupMenu script, which executes after the menu has been created, to add a function to the File menu to call that utility. Because we use an OOP menu, adding items to the menu is easy; here's the code for that script:

```
lparameters toApplication as SQApplication, toMenu
local loBar
loBar = toMenu.FilePad.AddBar('FileImportFF')
with loBar
   .cBarPosition    = 'before FileExportReport'
   .cPictureFile    = 'importxpsmall.bmp'
   .cCaption        = 'Import ACCPAC Report &Writer Reports...'
   .cStatusBarText  = 'Import reports from the ACCPAC Report Writer 6.5 or earlier'
   .cOnClickCommand = 'ImportFF()'
   .cSkipFor        = "type('_screen.ActiveForm.lCanImport') <> 'L' or " + ;
      "not _screen.ActiveForm.lCanImport"
endwith
toMenu.FilePad.Show()
```

Notice this script is passed references to the Stonefield Query application object and to the menu object. Of course, it also has access to anything within the application, including the main form (_screen.ActiveForm). When the user chooses this function, it calls another script, ImportFF, to do the conversion.

Like most accounting applications, Sage Pro consists of several modules, such as General Ledger, Accounts Receivable, and Inventory Control. Not all modules may be available; for example, a law firm likely wouldn't purchase the Inventory Control module. Stonefield Query should only show tables the user can actually report on, so it should remove those that belong to modules that aren't installed. The DataEngine.AfterDataGroupsLoaded event script, called after the data dictionary has been set up, handles that. It opens the Sage Pro system table containing the names of the installed modules and removes from the data dictionary those tables the user shouldn't see.

Since SQL Server doesn't have the concept of a blank date, dates that have no value (such as the date an item is shipped if it hasn't been shipped yet) are assigned the value 01/01/1900. Users don't like that; they'd rather see a blank in the report. The DataEngine.AfterPerformQuery event script (you can likely guess when it's called) uses this code to replace 01/01/1900 with a blank date for all Date and DateTime fields in the cursor retrieved for a report:

```
ldDate = date(1900, 1, 1)
```

```
ltDate = dtot(ldDate)
for lnI = 1 to afields(laFields)
   lcField = laFields[lnI, 1]
   lcType  = laFields[lnI, 2]
   do case
      case lcType = 'D'
         replace &lcField with {/} for &lcField = ldDate
      case lcType = 'T'
         replace &lcField with {/:} for &lcField = ltDate
   endcase
next lnI
```

There are lots of other scripts that customize Stonefield Query for Sage Pro to work the way users expect, but these examples should give you some ideas of the capabilities of plug-ins.

## Summary

Plug-ins can provide many benefits to your applications: extending or altering the functionality of the application, deploying new features without installing a new build, and creating customer-specific versions of an application without maintaining different code bases or many sets of CASE statements. As I've shown in this document, it isn't difficult to implement plug-in support: select an architecture that suits your needs and start coding!

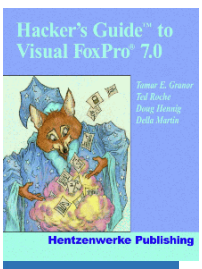
## Biography

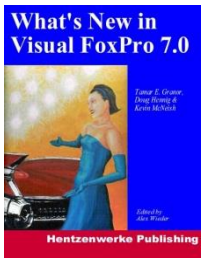
Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace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

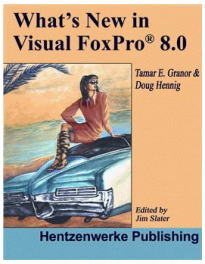
Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of *The Hacker's Guide 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 and Southwest Xbase++ conferences (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 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).

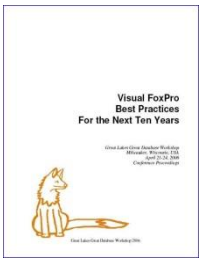
 

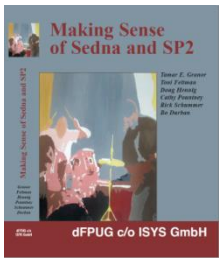

Hacker's Guide™ to Visual FoxPro® 7.0
Hentzenwerke Publishing

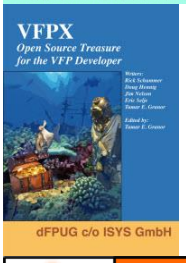

What's New in Visual FoxPro 7.0
Hentzenwerke Publishing


What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Hentzenwerke Publishing


What's New in Nine:
Visual FoxPro's Latest Hits
Hentzenwerke Publishing


Visual FoxPro
Best Practices
For the Next Ten Years


Making Sense of Sedna and SP2
dFPUG c/o ISYS GmbH


VFPX
Open Source Treasure for the VFP Developer
dFPUG c/o ISYS GmbH


stonefield
Database
TOOLKIT


stonefield
Query


MVP
Microsoft®
Most Valuable
Professional


VFPx


FoxRockX