

# Customizing VFP's Tools

*Session ???*

*Tamar E. Granor, Ph.D.*

*Voice: 215-635-1958*

*Email: [tamar\\_granor@compuserve.com](mailto:tamar_granor@compuserve.com)*

## **Overview**

Many of Visual FoxPro's tools, such as the Class Browser, the Builder system and the Coverage Profiler, are extensible. This session looks at the architecture of some of VFP's tools and demonstrates how to "have it your way." Familiarity with VFP and some experience using the tools is assumed.

## Tools and More Tools

Visual FoxPro includes a large collection of tools. Some, like the Form and Class Designers, are built right into the product. But many others are written in Visual FoxPro – Microsoft refers to these collectively as "Xbase tools" (because they're written in FoxPro, that is, Xbase, rather than C). The Xbase tools include the Class Browser and its alter ego, the Component Gallery; the Builders and Wizards; the Coverage Profiler; and, in VFP 7, the IntelliSense Manager; the Task List Manager; and the Object Browser.

## Replacing Xbase tools

VFP has a system variable corresponding to each of the Xbase tools. The variable points to the program to run for that tool. For example, the `_Browser` variable points to "Browser.APP" in the VFP home directory. Table 1 shows the list of Xbase tools, their corresponding system variables and the default values of those variables. (Note that there are some other system variables that point to programs, but those routines either don't fit the definition "tool," or aren't written in Xbase.)

**Table 1 FoxPro's Xbase tools –Each Xbase tool has a system variable that points to the code to run.**

Tool	System Variable	Default value
Builders	<code>_BUILDER</code>	<code>HOME() + "Builder.App"</code>
Class Browser	<code>_BROWSER</code>	<code>HOME() + "Browser.App"</code>
Component Gallery	<code>_GALLERY</code>	<code>HOME() + "Gallery.App"</code>
Coverage Profiler	<code>_COVERAGE</code>	<code>HOME() + "Coverage.App"</code>
IntelliSense Manager	<code>_CODESENSE</code>	<code>HOME() + "FoxCode.App"</code>
Object Browser	<code>_OBJECTBROWSE R</code>	<code>HOME() + "ObjectBrowser.App"</code>
Task List Manager	<code>_TASKLIST</code>	<code>HOME() + "TaskList.App"</code>
Wizards	<code>_WIZARD</code>	<code>HOME() + "Wizard.App"</code>

The system variables, in fact, provide an alternative way to run the various tools. Rather than choosing a tool from the menu or toolbar, you can also run it by DOing the corresponding variable. For example, to start the Coverage Profiler, you can issue this command:

```
DO (_COVERAGE)
```

One way to customize a tool is to replace it entirely and set the appropriate variable to point to your replacement. That's the strategy used by the GENMENUX menu generator wrapper program. To use it, you set `_GENMENU` to point to `GenMenuX.PRG`.

One of the built-in tools can also be replaced – the Expression Builder. Set `_GETEXPR` to point to the program you'd rather use and both the IDE and calls to `GETEXPR` use the program you specify.

In VFP 6 and later, the source code for the Xbase tools comes with VFP. (Unzip XSource.ZIP in the Tools directory.) So another customization strategy is to modify or subclass the existing code, if it doesn't do exactly what you want.

But the truth is that most of us would find either replacing or modifying code for any of the tools a daunting task.

## Open Architecture

Fortunately, there are other ways. Several of the tools (the Class Browser/Component Gallery, the Coverage Profiler, and the Object Browser) accept "add-ins," pieces of code you write and then can call from specified points in the tool.

The Builder and Wizard system is table-driven. To add a builder or wizard, you add a record to the appropriate driver table.

While the IntelliSense Manager itself can't be customized, IntelliSense in VFP is itself table-driven. (The table is referenced through the `_FOXCODE` system variable.) The IntelliSense Manager provides several ways of changing IntelliSense behavior.

The Task List Manager is also table-driven, with the table referenced through the `_FOXTASK` system variable. In addition, its interface provides the ability to add custom fields to tasks and have any or all of those fields displayed in the Task Manager. The Task List's object model also makes it possible to add and manipulate tasks programmatically.

Even VFP's built-in tools, like the Project Manager and most of the Designers, have open architecture. Each of them stores their data in a table that uses special extensions. Table 2 shows the file extensions used for each tool.

**Table 2 Designer Open Architecture – Most of VFP's built-in tools store their data in a table with special extensions.**

Tool	Table (DBF)	Memo fields (FPT)
Class Designer	VCX	VCT
Form Designer	SCX	SCT
Label Designer	LBX	LBT
Menu Designer	MNX	MNT
Project Manager	PJX	PJT
Report Designer	FRX	FRT

It's possible to directly manipulate data from these tools by modifying the tables directly. In addition, projects offer the ProjectHook class, which has methods that fire when various events occur on the project. For example, you can write code to run whenever a file in the project is modified.

## **Add-ins**

An add-in is a program hooked into another program. It generally manipulates the properties of the host program, and may use the host's methods to do so. An add-in performs a task not built into the host program.

The Class Browser, Component Gallery, Coverage Profiler and Object Browser all support add-ins. However, the techniques for specifying and using them vary with the tool.

The Class Browser and Component Gallery (which are really two faces for the same tool) have an AddIn method that lets you register and unregister add-ins. Add-ins in the CB/CG can either be added to a menu of add-ins or hooked to a specific event, such as the Click of a particular button.

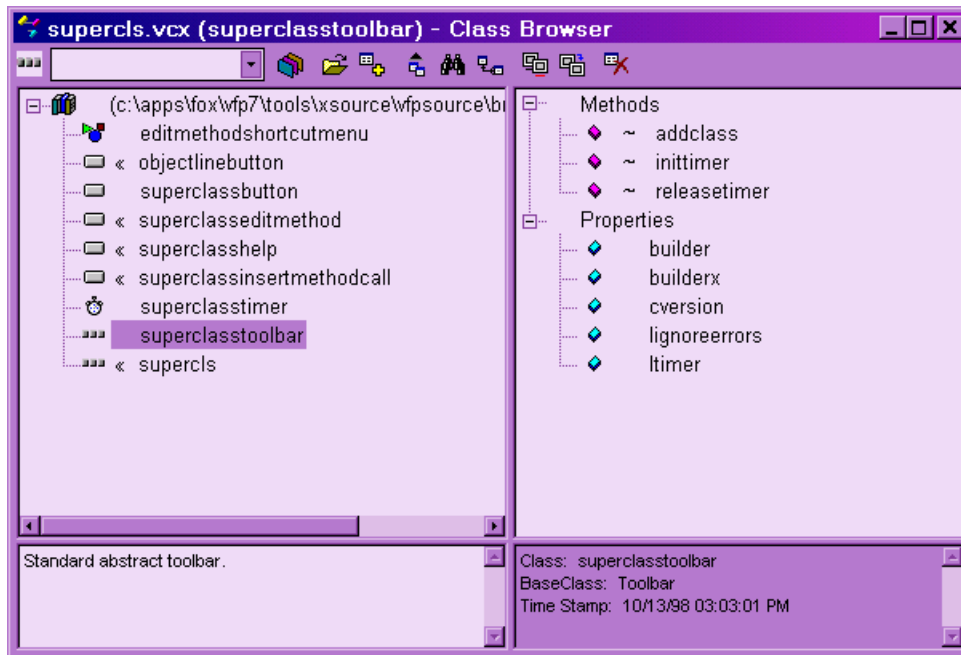
The Coverage Profiler has an Add-ins item on its context menu and toolbar. When you choose an add-in to run, you can also specify that it should be registered and thus made available for future Profiler sessions. In addition, you can use the AddTool method of the Coverage Profiler's main form to add a button or other control that runs an add-in.

The Object Browser also includes an Add-ins item on its context menu that offers the opportunity to install an add-in. Add-ins can also be installed and removed using the Add-ins page of the Options dialog.

Add-ins for the Class Browser, Component Gallery and Coverage Profiler are usually programs, but can be any directly executable piece of code (such as a form, an APP or an EXE). Object Browser add-ins are classes, usually VCX-based, and work best when subclassed from the `_baseaddin` class provided.

## **Adding to the Class Browser**

The Class Browser provides an easy way to look inside class libraries to see the relationships among classes, as well as the structure of individual classes. It's a remarkably capable tool. (For thorough coverage of the Class Browser's abilities, see the Egger book listed in the "Resources" section.) Figure 1 shows the Class Browser listing the classes for the SuperClass toolbar.



**Figure 1 Using the Class Browser – The Class Browser lists classes and their members.**

However, there are some tasks you might want to do with your class libraries that aren't included. To give you the ability to add such tasks, the Class Browser includes add-in capability.

Whenever an instance of the Class Browser is running, a public variable `_oBrowser` is created, if necessary, and given an object reference to the active Class Browser object. This reference is useful for registering add-ins, as well as for checking the state of the Class Browser.

## Registering Class Browser add-ins

Attaching an add-in to the Class Browser is easy. When you call the Class Browser's `AddIn` method, the add-in is added to the Class Browser's registration table (`Browser.DBF` in the VFP home directory). From that point on, the add-in is available. Add-ins do not need to be re-registered each time you run VFP or the Class Browser.

The `AddIn` method has six parameters, but you'll usually use only the first three. The first two are required. Here's the syntax:

```
_oBrowser.AddIn( cName , cProgram [, cMethod ])
```

`cName` is the name of the add-in, which appears on the add-in menu, if the `cMethod` parameter is omitted. `cProgram` is the program to run when the add-in is called. The optional `cMethod` parameter lets you specify that the add-in should run automatically when a particular Class Browser event fires.

For example, to register an add-in called `MyAddIn`, which is implemented by `MyAddIn.PRG` in the folder `c:\apps\utils`, you'd issue:

```
_oBrowser.AddIn( "MyAddIn", "c:\apps\utils\MyAddIn.PRG")
```

To register the same add-in, but set it up to fire whenever the Class Browser's Activate method fires, register it like this:

```
_oBrowser.AddIn( "MyAddIn", "c:\apps\utils\MyAddIn.PRG", "Activate")
```

To register an add-in to fire based on an event of one of the Class Browser's controls, include the control name in the event name. For example, this call indicates that the add-in should fire when the user right-clicks on the Add button:

```
_oBrowser.AddIn( "MyAddIn", "c:\apps\utils\MyAddIn.PRG", ;  
  "cmdAdd.RightClick")
```

A given add-in can only be fired in one way, so if we issued the three calls to AddIn shown here in sequence, the add-in would be called only when the user right-clicks on the Add button.

Finally, to unregister an add-in, call the AddIn method, but pass .null. for the cProgram parameter. For example, to unregister the add-in registered above, issue:

```
_oBrowser.AddIn( "MyAddIn", .null. )
```

## Structuring a Class Browser add-in

When a Class Browser add-in is called, no matter what triggers it, it receives a single parameter: an object reference to the Class Browser itself. This provides the add-in with the ability to access the Browser object's properties and methods without using the \_oBrowser variable. This is important because it's possible to have multiple instances of the Class Browser open, so you don't know which instance the \_oBrowser variable points to.

The add-in itself can contain any code at all. For testing purposes, you may want to create a simple add-in, like the following:

```
LPARAMETERS oBrowser  
  
WAIT WINDOW "This is the add-in"  
RETURN
```

However, a useful add-in is likely to work with the Class Browser's object model. Its properties and methods are documented in the VFP Help file (see the topics "Class Browser Properties" and "Class Browser Methods"). Some of the key PEMs you're likely to work with are listed in Table 3.

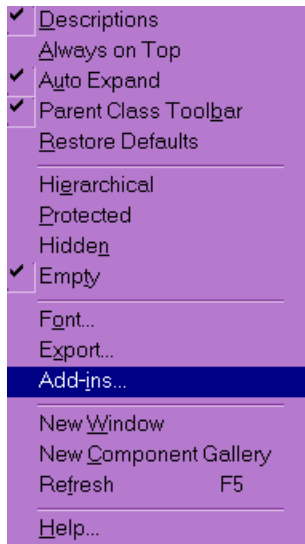
**Table 3 Class Browser PEMs – The Class Browser has a rich object model. These properties and methods are likely to be used in many add-ins. See Help for the complete list.**

Name	Property/Method	Purpose
aClassList	Property	An array property containing a list of all the classes and forms in the class list, with one row for each.
AddFile	Method	Opens an existing class library or form, without closing the files that are already open.

Name	Property/Method	Purpose
aFiles	Property	An array property containing a list of all the files currently open in the Class Browser, with one row for each file.
cClass	Property	Contains the name of the currently selected class in the class list.
cClassLibrary	Property	Contains the name of the class library for the currently selected class in the class list.
cFileName	Property	The full path to the file containing the item currently selected in the class list.
ExportClass	Method	Generates and, optionally, displays code for the class, form or file selected in the class list.
ModifyClass	Method	Opens the currently selected class in the Class Designer, optionally opening the method editor to a specified method.
NewFile	Method	Creates a new class library and, optionally, opens it in the Class Browser.
OpenFile	Method	Opens an existing class library or form in the Class Browser, closing the files that are already open.
RedefineClass	Method	Changes the parent class of the currently selected class.
RemoveClass	Method	Removes the selected class from its class library.
RenameClass	Method	Renames the selected class in the class list.
SeekClass	Method	Moves the class list pointer to the specified class.
SeekMember	Method	Moves the member list pointer to the specified member.
SeekParentClass	Method	Moves the class list pointer to the parent class of the currently selected class, opening the library containing the parent class, if necessary.

## Using a Class Browser add-in

Running a Class Browser add-in is much easier than writing it or registering it. If the add-in is hooked to a Class Browser event, simply triggering that event runs the add-in. If the add-in was placed on the Class Browser menu, right-click anywhere on the Browser, except in the areas that hold the class list and member list. The context menu that appears includes an Add-ins... item. (Figure 2 shows the Class Browser's context menu.) Choose that item and a list of menu add-ins appears. Choose the add-in you want to run.



**Figure 2 Running an add-in – To run an add-in that isn't attached to a Class Browser event, choose Add-ins from the context menu.**

## **An example: Subclassing a class library**

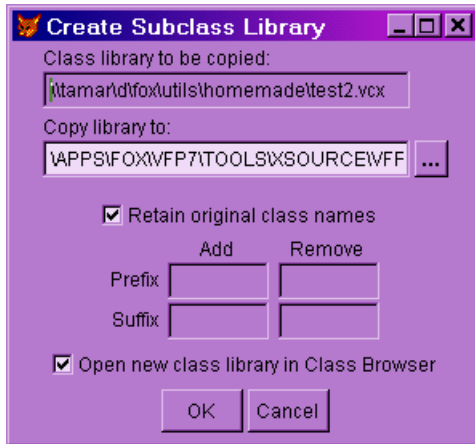
Whenever I start a new project, I want to subclass each of my personal base classes, to provide a starting point for the project. This task sounds like something that should be easy in the Class Browser, but in fact, doing it there is quite a tedious, manual operation. Clearly, an automated process is called for.

The add-in should prompt for a name and location for the destination class library, then make a copy of every class from the currently selected class library (the source), placing the copies in the destination library. Before moving to code, we need to consider some other issues for the add-in.

Many people use prefixes or suffixes on class names to indicate their position in the class hierarchy (distinguishing abstract classes from concrete) or to specify the client or project a class is meant for. So, the add-in needs a way to change the prefix and/or suffix of the class name.

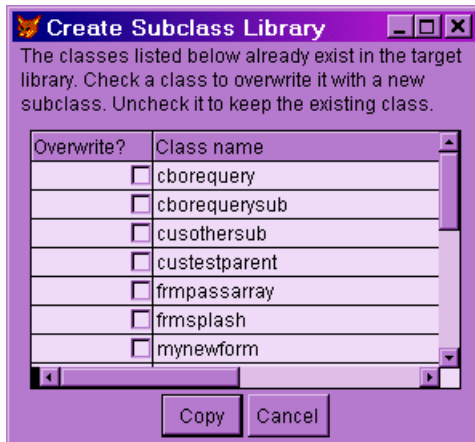
Figure 3 shows the main interface to the Create Sublibrary add-in. It allows the user to specify the destination library, to change prefixes and suffixes, and to indicate whether the new library should be opened in the Class Browser after the task is done.





**Figure 3 The Create Sublibrary add-in – This add-in subclasses every class in the current class library. The user specifies the destination and deals with naming conventions.**

It's possible that some of the classes to be copied already exist in the destination class library. (For example, perhaps we've added some new classes to the base class library and want to update the various copies.) The best solution is to give the user control over what happens in this case. Figure 4 shows the approach used.



**Figure 4 Handling duplication – If a class in the source class library already exists in the destination class library, the user can choose whether to overwrite it.**

The add-in consists of a main program that calls a form defined in code. While an SCX-based form can be used as an add-in, a coded form allows the add-in to be distributed as a single file. The main program does a lot of error checking. If all is well, it instantiates and shows the form, passing an object reference to the Class Browser.

The form contains a tabless pageframe with two pages (shown in Figures 3 and 4). It creates its controls from a number of other classes defined in the same file. The form uses two cursors created in the Load method. Classes contains the list of classes in the library to be copied (the source), while Preexists holds the list of classes in the destination library that duplicate classes to be copied. The key methods are ValidateCopy, DetectDups and CopyLibrary.

ValidateCopy drives the whole copy process-it's called by the OK button on the first page (Figure 3). First, it creates the new name for each class by removing the old prefix and suffix and adding the new ones. Then it updates the Classes cursor with the new name. When all the names have been transformed, DetectDups is called to create a list of duplicate class names. If any matches are found, the second page (Figure 4) is activated and the method ends. If no class names are duplicated, CopyLibrary is called to perform the copy. Here's the code for ValidateCopy:

```

PROCEDURE validatecopy
* Creates the new class names and posts them to the Classes Cursor
* Calls either the copy method or the page with the grid for resolving
* the duplicate names
LOCAL lcNewClassName, lcNewPreFix, lcOldPrefix, lcNewSuffix,
LOCAL lcOldSuffix

SELECT Classes
WITH Thisform.pageframe1.page1
SCAN
  * Get the old class name in the existing library
  lcNewClassName = ALLTRIM(OldClassName)
  IF .chkRetainNames.Value
    * If no mods were requested, post the old name as the new
    * and get the next one
    REPLACE NewClassName WITH lcNewClassName
  LOOP
ENDIF

  * Get the name change requests
  lcNewPrefix = ALLTRIM(.txtNewPrefix.Value)
  lcNewSuffix = ALLTRIM(.txtNewSuffix.Value)
  lcOldPrefix = ALLTRIM(.txtOldPrefix.Value)
  lcOldSuffix = ALLTRIM(.txtOldSuffix.Value)

  IF NOT EMPTY( lcOldPreFix )
    * If a prefix is to be removed
    IF UPPER(LEFT(lcNewClassName,LEN(lcOldPrefix))) = ;
      UPPER(lcOldPrefix)
      * If that prefix is on this class name, remove it
      lcNewClassName = SUBSTR(lcNewClassName,LEN(lcOldPrefix)+1)
    ENDIF
  ENDIF

  IF NOT EMPTY( lcNewPreFix )
    * If a prefix is to be added, add it
    lcNewClassName = ALLTRIM(lcNewPrefix) + lcNewClassName
  ENDIF

  IF NOT EMPTY( lcOldSufFix )
    * If a suffix is to be removed
    IF UPPER(RIGHT(lcNewClassName,LEN(lcOldSuffix))) = ;
      UPPER(lcOldSuffix)
      * If that suffix is on this class name, remove it
      lcNewClassName = SUBSTR(lcNewClassName,1,;
        LEN(lcNewClassName)-LEN(lcOldSuffix))
    ENDIF
  ENDIF

  IF NOT EMPTY( lcNewSufFix )
    * If a suffix is to be added, add it
    lcNewClassName = lcNewClassName + ALLTRIM(lcNewSuffix)

```

```

ENDIF

* Post the new name
REPLACE NewClassName WITH lcNewClassName
ENDSCAN
ENDWITH

* Detect any classes that may preexist in the destination library
ThisForm.DetectDups()

* Did any get found
LOCATE FOR Preexists
IF FOUND()
* If yes, display the grid for overwrites
ThisForm.Pageframe1.ActivePage = 2
* Since the grid page does the copying if requested get out of here
RETURN
ENDIF

* Do the actual copying of the classes
ThisForm.CopyLibrary()

RETURN
ENDPROC

```

**DetectDups** checks the new names against the list of classes in the destination library and adds a record to **Preexists** for each match it finds. Here's the code:

```

PROCEDURE detectdups
* Searches target library for existing classes with duplicate names
LOCAL nExisting, aExistingClasses[1], nMatches, aMatches[1], nMatchPos,
LOCAL lcNewLib, lcOldLib

lcOldLib = ALLTRIM(ThisForm.Pageframe1.Pagel.txtSourceLibrary.Value)
lcNewLib = ALLTRIM( ;
ThisForm.Pageframe1.Pagel.txtDestinationLibrary.Value)

IF NOT FILE( lcNewLib )
* The target library does not exist, then there is nothing to do here
RETURN
ENDIF

SELECT Classes
* Get a list of classes in the target library
nExisting = AVCXClasses( aExistingClasses, lcNewLib )

IF nExisting > 0
* If there are any, compare existing classes to classes in
* source library
nMatches = 0
FOR nClass = 1 TO nExisting
* Search the source new class names for a match
LOCATE FOR UPPER( TRIM( NewClassName ) ) = ;
UPPER( TRIM( aExistingClasses[ nClass, 1 ] ) )
IF FOUND()
* If we found a matching name, mark the Preexists Field of the
* Classes cursor
REPLACE Preexists WITH .T.
INSERT INTO Preexists ( NewClassName ) ;
VALUES ( Classes.NewClassName )
ENDIF
ENDIF

```

```

    ENDFOR
  ENDIF

  RETURN
ENDPROC

```

CopyLibrary is called, as explained above, from the ValidateCopy method. It's also called from the Copy button on the second page of the form. The method creates the destination class library, if it doesn't already exist, then subclasses all the classes, except those marked by the user not to be overwritten.

While the Browser has a method called NewClass, unfortunately, it doesn't let you store the new class in a library other than the one currently selected. It also can only create classes based on the VFP base classes and the classes in the current library. So, creating the subclasses isn't as simple as calling NewClass for each class in the class library.

CopyLibrary uses the CREATE CLASS command instead, as it has all the flexibility needed. However, it doesn't have a NOSHOW clause to create the new class silently. CREATE CLASS always opens the Class Designer, so CopyLibrary uses KEYBOARD to issue the appropriate keystrokes to save the new class and close the designer.

Here's the code for CopyLibrary:

```

PROCEDURE copylibrary
LOCAL lcNewLibrary, lcOldLibrary, lcNewClass, lcOldClass

* Get name of the target library
lcNewLibrary = ALLTRIM( ;
  Thisform.PageFrame1.Page1.txtDestinationLibrary.Value)

* Get name of the source library
lcOldLibrary = ALLTRIM( ;
  Thisform.PageFrame1.Page1.txtSourceLibrary.Value)

* Check for any preexisting classes
SELECT Preexists
IF RECCOUNT() > 0
  * If any preexisting classes mark them according to the users choice
  SCAN FOR CopyOver
    SELECT Classes
      LOCATE FOR NewClassName = Preexists.NewClassName
      REPLACE CopyOver WITH .T.
    ENDSCAN
ENDIF

SELECT Classes

IF NOT FILE(lcNewLibrary)
  * If the target library does not exist create it
  CREATE CLASSLIB (lcNewLibrary)
ENDIF

* Process the classes in the source library
SCAN
  IF NOT Classes.Preexists OR (Classes.Preexists AND Classes.CopyOver)
    * If the class is new to the target or it is marked to be
    * an over write
    IF Classes.Preexists

```

```

* If this is an over write remove the existing class
* from the target
REMOVE CLASS (ALLTRIM(Classes.NewClassName)) OF (lcNewLibrary)
ENDIF

* Set up to shut down the Class Designer
KEYBOARD "{CTRL+F4}Y"

* Create the class
CREATE CLASS (ALLTRIM(Classes.NewClassName)) OF (lcNewLibrary) ;
AS (ALLTRIM(Classes.OldClassName)) FROM (lcOldLibrary)
ENDIF
ENDSCAN

* Now open this library if requested
IF THISFORM.Pageframe1.Page1.chkOpenLib.Value
    This.oBrowser.AddFile(lower(lcNewLibrary))
ENDIF

RETURN
ENDPROC

```

The complete add-in is included in the materials for this session as NewLib.PRG. NLReadme.TXT is a readme file for this add-in.

## Enhancing the Component Gallery

The Component Gallery is another face for the same application as the Class Browser. (The main program of Gallery.APP calls Browser.APP, passing a parameter to indicate that it should open in "gallery mode.")

This means that the technique for registering add-ins is the same – call the AddIn method – and that Component Gallery add-ins also receive a reference to the calling Browser/Gallery instance.

Unfortunately, the resemblance ends there. While the Class Browser PEM's are well-documented and several articles have been written about extending the Class Browser, the Component Gallery's PEM's are totally undocumented (there aren't even descriptions of them in the main Browser form) and no such articles exist.

The best way to figure out what Component Gallery PEM's are relevant to a particular task is to open the Component Gallery, then explore its members in the Debugger and Command Window, using the \_oBrowser reference. Table 4 shows some of the key properties for the Component Gallery.

**Table 4 Component Gallery properties – Unfortunately, these aren't documented anywhere. This list shows some of the key items.**

Name	Purpose
aFolderList	An array property containing one member for each catalog or folder available. Each item is an object reference to a _folder object.
aItem	An array property containing one member for each item of each folder that's been examined during this session. Each array element is either an object reference to an object based on (or subclassed from) the _folder or _item

	class, or contains a delimited list with key information about the item.
cCatalog	The currently chosen catalog.
nFolderCount	The number of folders in aFolderList.
nFolderListIndex	The position of the currently selected catalog in aFolderList.
nItemCount	The number of items in aItemList.
nItemListIndex	The position of the currently selected item within the current folder.
oCatalog	An object reference to the currently selected catalog.
oFolder	An object reference to the currently selected folder.
oItem	An object reference to the currently selected item.

The Component Gallery also offers an entirely different approach to extension. You can create your own item types that can be used in the Gallery just like the types provided. The Egger book listed in the Resources section explains this approach in detail.

## Extending the Coverage Profiler

The Coverage Profiler takes a coverage log (created by issuing SET COVERAGE TO <filename> or by turning coverage logging on in the Debugger) and turns it into meaningful information. It offers information about which lines were executed and which were not, as well as timing information for those lines which were executed.

The Coverage Profiler can be enhanced in a variety of ways. Its design separates interface from implementation - there's a coverage engine class and a coverage interface class. Each can be subclassed independently and it's also possible to create an entirely new user interface for the coverage engine. The Nicholls article listed in the Resources section discusses a variety of subclassing options.

The COV\_TUNE.H include file (in the VFPSrc\Source\Coverage directory, when you unzip the provided source) allows you to fine-tune various Coverage Profiler behavior. It's well commented to show you what effects your changes will have. Of course, once you make changes, you'll need to rebuild COVERAGE.APP.

The simplest way to extend the Coverage Profiler, though, is with add-ins. As with the Class Browser/Component Gallery, it's easy to attach add-ins to the Coverage Profiler. The Coverage Profiler's toolbar and context menu both contain an Add-ins item. When you choose it, a dialog (shown in Figure 5) appears. The dialog allows you to locate and execute an add-in. The dropdown contains a list of all add-ins executed in this Coverage Profiler session.



**Figure 5 Running a Coverage Profiler add-in – This dialog allows you to find, run, and register Coverage Profiler add-ins.**

The checkbox in the dialog lets you register an add-in. Once registered, the add-in appears in the dropdown every time you use the Coverage Profiler. By default, the list of registered add-ins is stored in the Registry (in the key HKEY\_CURRENT\_USER\Software\Microsoft\Visual FoxPro\<version>\Coverage). While there's no mechanism provided for unregistering an add-in, it is possible to delete the relevant Registry key.

Add-ins receive as a parameter an object reference to the Coverage Profiler engine object. There's also a public variable, `_oCoverage`, that points to this object.

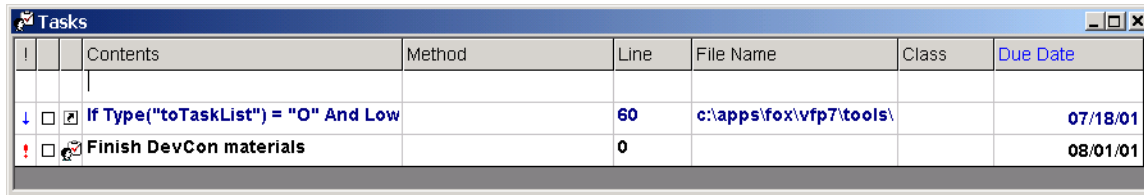
The object model for the Coverage Profiler's engine is well documented in the VFP Help. (See the Coverage Engine Object topic.) Two properties you're likely to use in almost any add-in are `cSourceAlias`, which contains the alias for a cursor containing the parsed coverage log, and `cTargetAlias`, which contains the alias for a cursor containing processed coverage data.

Many Coverage Profiler add-ins won't need to deal with any other properties, but will simply process the data in one or both cursors. An example of a Coverage Profiler add-in that processes the raw log data follows the discussion of the Task List.

The Coverage Profiler makes it easy to add to its own interface. The `cov_maindialog_standard` form class, which is used for the Coverage Profiler's main form, has an `AddTool` method. This method accepts a class name as parameter and adds an instance of that class to the Coverage Profiler form.

## Customizing the Task List

VFP 7 offers a new tool, the Task List, which contains shortcuts to lines of code and any other tasks a developer chooses to add. Tasks can be added through the Task List interface (shown in Figure 6) or by setting shortcuts in code editing windows.



	Contents	Method	Line	File Name	Class	Due Date
↓	<input type="checkbox"/> If Type("toTaskList") = "O" And Low		60	c:\apps\fox\vfp7\tools\		07/18/01
!	<input type="checkbox"/> Finish DevCon materials		0			08/01/01

**Figure 6 The Task List – This tool, new in VFP 7, lets you manage reminders and pieces of code that need attention.**

As with the other Xbase tools, the Task List's source code is provided. VFP 7 also includes two documents that describe the Task List's object model and specifications. However, there's no add-in mechanism and the object model is not documented in the Help file.

So why include this tool in a session on extending VFP's tools? Because, despite these shortcomings, there are a few things you can do with the Task List without subclassing or replacing its code.

As with the other tools, a public variable (`_oTaskList`) containing object reference to the tool is created when you open it.

The tool provides for a list of custom fields, in addition to the built-in list. Right-click and choose Options. There you can specify or create a "user-defined column table." This table must include a 10-character UniqueID field (used to link the custom fields to the standard ones), but otherwise you can specify whatever fields you want.

Since the task data is stored in a table, it's possible to add tasks programmatically. However, it's better to work through the Task List's object model than just use straight VFP code.

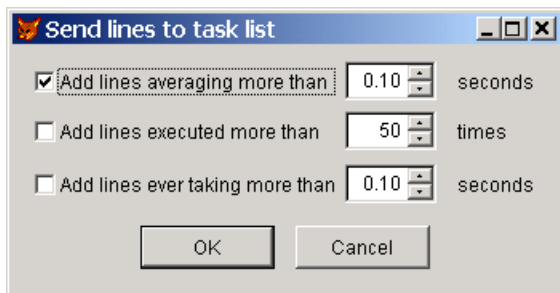
The technique for adding a new task is a little roundabout (but similar to the way you add items to Outlook). First, you request an empty task object using the GetTaskObject method. When you've filled it in, you add it to the list using the AddTask method. The task object has a property for each field, whether built-in or custom. The property name is the field name preceded by an underscore ("\_"). For example, the Contents field is represented by an \_Contents property.

Editing tasks uses a similar paradigm. The GetTask method accepts the unique id of a task and returns a task object. The UpdateTask method accepts a task object and updates that task in the table.

## An example: A Coverage Profiler add-in to add tasks to the Task List

Generally, once you've run a coverage log and looked at the results, you have a number of items for your "to do" list. So, an obvious enhancement to the Coverage Profiler is an automated way of adding tasks based on coverage information.

This add-in decides what tasks to create based on the amount of time a line took to execute and the number of times it was executed. A form (see Figure 7) lets you choose the criteria for sending lines to the task list. Behind the scenes, the Coverage Profiler and Task List objects are used to figure out which lines are affected and to create a task for each.



**Figure 7 Creating tasks automatically – This Coverage Profiler add-in adds a task for each line in the log that meets the specified criteria.**

This add-in is implemented as a single form (CovToTask.SCX in the session materials), with a number of custom properties and methods. The main processing method is ProcessLog, partially shown here. (Two additional IF blocks handle the other two conditions for adding tasks.)

```
LOCAL lRetVal  
  
This.computelinetimes()  
  
IF This.laddforavgtime
```



```

SELECT * ;
    FROM LineTotal ;
    WHERE nAvgTime > This.navgtime ;
    INTO CURSOR AboveAverage

lSuccess = .T.
SCAN WHILE lSuccess
    * Bail out if the addition fails
    lSuccess = This.addtask( ObjClass, Executing, ProcLine, ;
        HostFile, nAvgTime, "Average Time")
ENDSCAN

USE IN AboveAverage
lRetVal = lSuccess
ENDIF

* Code to process the log for the other two conditions
* omitted for space

```

The ComputeLineTimes method contains a single query that consolidates information from the coverage log. Note the use of the cSourceAlias property of the Coverage object:

```

* Total time, number of times, average time,
* and maximum time for each line
SELECT objclass, executing, procline, hostfile, ;
    sum(duration) AS nTotalTime, count(*) AS nTimes, ;
    avg(duration) AS nAvgTime, MAX(duration) AS nMaxTime ;
FROM (This.ocoverage.cSourceAlias) ;
GROUP BY objclass, executing, procline, hostfile ;
INTO CURSOR linetotal

```

The AddTask method ensures that we have a reference to the Task List, then creates and adds a task. (Assertions for parameter checking are omitted here.)

```

* Add a task to the task list

LPARAMETERS cObjClass, cExecuting, nProcline, cHostfile, nTime, cReason
* First four params come right from coverage log
* Fifth parameter is measured time
* Last parameter is test failed

* Assertions omitted for space

LOCAL oTask

IF This.Gettasklist()

WITH This.oTasklist
    oTask = .GetTaskObject()
    WITH oTask
        ._Type = "S"
        IF UPPER(JUSTEXT(cHostFile)) <> "SCX"
            ._Class = ALLTRIM(cObjClass)
        ELSE
            ._Class = ""
        ENDIF
        IF INLIST(UPPER(JUSTEXT(cHostFile)), "SCX", "VCX")
            ._Method = ALLTRIM(cExecuting)
        ELSE
            ._Method = ""
        ENDIF
    ENDWITH
ENDWITH

```

```

ENDIF
  ._Line = nProcLine
  ._FileName = ALLTRIM(cHostFile)
  ._Contents = This.FindCodeLine( cHostFile, cExecuting, ;
    nProcLine )
  ._cReason = ALLTRIM(cReason) + " " + TRANSFORM( nTime )
ENDWITH
  .AddTask( oTask )
ENDWITH
  lRetVal = .T.
ELSE
  lRetVal = .F.
ENDIF

RETURN lRetVal

```

AddTask uses two additional methods of the class. GetTaskList gets a reference to the task list object, starting the tool, if necessary. It also ensures that the user-defined column table exists and includes a column called cReason. (To do so, it uses an instance of another class, cusSaveTable, that includes methods to store information about all open instances of a table, and to reopen a table in all data sessions where it was previously open.) FindCodeLine returns the actual code for a specified line-it's fairly standard VFP string-handling code.

This add-in reveals a bug in VFP's internal handling of task shortcuts. If the case in the Method field doesn't exactly match VFP's "native" case, the shortcut icon doesn't appear when the file is opened. ("Native" case means camel-case for method names and defined case for controls.) However, the file can still be opened to the right method with the cursor on the right line. Microsoft is aware of the bug.

## Building a Builder

The Builder technology introduced in VFP 3 may be one of the most underused facilities in the product. Because almost all the builders included with VFP are just wizard-like formatting and data set-up tools, very few developers realize the potential of builders. In fact, builders let you modify forms, controls and classes under construction. Builders don't have to have a user interface, but can perform all their actions behind the scenes.

A builder can be as simple as a few lines that grabs an object reference to something from a designer and changes a few properties, or it can be far more complex.

Builders can be run in a variety of ways, as well, and do not have to be hooked into the built-in Builder mechanism. You can run a builder from the Command Window or a menu item, if you choose. However, the built-in mechanism makes it easy to make a builder available whenever it's appropriate.

## Registering builders

All builders registered in the Builder.DBF table located in the Wizards subdirectory of VFP are available through the built-in mechanism. The structure of the Builder table is shown in Table 4. (It's also documented in the VFP Help file.) You don't generally need to specify the ClassLib, ClassName and Parms fields. It's sufficient to provide Name, Descript, Type and Program.

**Table 4 Registering a builder – To make your builder available through VFP's Builder mechanism, add a record to Builder.DBF containing these fields.**

Field name	Type	Content
Name	Character	The name for your builder. This name is displayed in the Builder Selection dialog, if there's more than one appropriate builder available.
Descript	Memo	The description of your builder. Displayed in the Description editbox of the Builder Selection dialog.
Bitmap	Memo	Currently unused
Type	Character	The control class to which your builder applies. Specify "ALL" for a builder that can be used on any type of control. Specify "MULTISELECT" for a builder that can be applied to multiple controls simultaneously.
Program	Memo	The name, including path, of the program or application to run for this builder.
ClassLib	Memo	The name, including path, of the class library containing the builder.
Class	Memo	The name of the class within ClassLib that constitutes the builder.
Parms	Memo	Parameters to pass to the builder.

You can create the necessary record manually, by opening the table and using INSERT INTO or BROWSE. However, a really handy strategy is to create a main program for your builder that registers it, if necessary, then runs the actual builder. (Thanks to Doug Hennig for this idea.) Here's an example (drawn from the builder discussed later in this section and included in the session materials):

```

=====
* Program:          BASEBUILDERMAIN.PRG
* Purpose:          Run the base class builder. Self-register, if
*                   necessary.
* Author:           Tamar E. Granor
* Copyright:        (c) 2001, Tamar E. Granor
* Last revision:    07/05/01
* Parameters:       As passed by the builder system
* Returns:          (None)
* Environment in:
* Environment out:
=====
* Main program for BaseBuilder

LPARAMETERS uP1, uP2, uP3, uP4, uP5, uP6, ;
              uP7, uP8, uP9, uP10, uP11, uP12
* Accept parameters passed by the builder system

#DEFINE ccMAIN      "BASEBUILDERMAIN"

LOCAL nOldSelect

* Self-register if called directly

```

```

IF PROGRAM(0) == ccMAIN
  nOldSelect = SELECT()
  SELECT 0
  USE HOME() + "Wizards\Builder" AGAIN
  LOCATE FOR UPPER(Name) = "BASE CLASS BUILDER"
  IF NOT FOUND()
    m.Name = "Base Class Builder"
    m.Descript = "Choose a class for base class controls"
    m.Type = "ALL"
    m.Program = SYS(16)
    INSERT INTO Builder FROM MEMVAR
  ENDIF

  USE IN Builder
  SELECT (nOldSelect)
ENDIF

* Run the actual builder
DO FORM ADDBS(JUSTPATH(SYS(16))) + "BaseBuilder" WITH ;
  uP1, uP2, uP3, uP4, uP5, uP6, ;
  uP7, uP8, uP9, uP10, uP11, uP12

RETURN

```

The program first checks to see whether it was run directly (as opposed to through the Builder mechanism). If so, it checks the Builder table to see whether this builder is already registered. If not, it registers it by inserting a record into table. After all that, it runs the program (a form, in this case) that constitutes the actual builder.

## Structuring a builder

The builder application passes up to 12 parameters to builder programs, so the main program or Init method of any registered builder must accept those parameters. Having done so, however, it's unlikely that you need to do anything with the parameters.

Most builders are meant to work on the currently selected object or objects. The ASelObj() function creates an array of object references to those objects, one per row. (The function is also capable of retrieving a reference to the container object of the selected objects, or to the data environment of the containing form.)

Given a reference to the selected object or objects, you can write code that reads or modifies properties and methods. Since builders operate at design-time, it's even possible to insert method code using the WriteMethod method. Using AddObject and RemoveObject, you can change the contents of the form or class. In VFP 6 and later, you can add properties to the objects using the AddProperty method.

Like the builders provided with VFP, your builders can display an interface that gives the user options. However, for some builder tasks, no interface may be needed. For example, you might write a builder that replaces all base class controls with your subclasses. (The example below is a variation on this theme.)

## Extending the builder system

VFP's builder mechanism has a hidden feature that can be especially useful in team development situations where one developer creates classes that other developers use. If a class has a Builder property, requesting a builder for that class runs the program specified in the Builder property. So the control developer can create both the control classes and builders that set up the controls properly. Other developers can drop controls on forms (or other classes), then right-click and choose Builder and be prompted to fill in the information necessary for the control to operate.

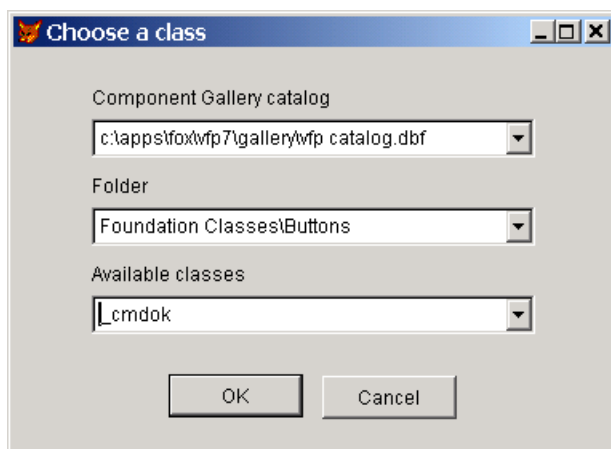
Ken Levy, tool developer extraordinaire, has extended the builder mechanism to make creating builders easier. The BuilderB and BuilderD technologies make it possible to create builders without starting from scratch each time. These approaches are well-documented in the Hennig white paper and FoxTalk articles listed in the Resources section.

## An example: Replacing base class objects

When dropping controls onto forms (or onto classes), it's easy to use a control from the wrong class. It's especially common to use a base class control instead of the one you really want. Once you've set properties and added code, changing the control is a pain. This builder (BaseBuilder.SCX in the session materials) solves that problem by letting you replace a control with any control derived from the same base class.

The builder form (shown in figure 8) has one custom property, oControl, which holds an object reference to the control being replaced. It also has three custom methods:

- About is a documentation method that describes the form.
- EnableControls handles enabling and disabling of controls on the form as changes are made.
- ReplaceControl performs the actual replacement of the selected control with the newly chosen control.



**Figure 8 The Base Class Builder – This builder lets you replace a control with another control derived from the same base class.**

Much of the work of the form is done in a custom control, cntCGClass (found in Utils.VCX in the session materials), which allows the user to choose a class from those in Component Gallery catalogs. This control uses three combos to list the catalogs, the folders within the currently chosen catalog and the controls of the right base class within the currently chosen folder. The control works directly with the Component Gallery's data tables.

Here's the code for the ReplaceControl method of the form:

```
* Replace the specified control with
* a control of the chosen class.

LOCAL oForm, oOriginalControl
LOCAL cClass, cClassLib, aOldProps[1]

cClass = This.cntcgclass.getchosenclass()
cClassLib = This.cntcgclass.GetChosenClasslib()

* First, get a reference to the containing form
oForm = This.oControl
DO WHILE UPPER(oForm.Baseclass) <> "FORM"
    oForm = oForm.Parent
ENDDO

* Save a reference to the original control
oOriginalControl = This.oControl

* Add the new control
cTempName = SYS(2015)
oForm.NewObject( cTempName, ;
                cClass, cClassLib )
oControl = EVALUATE( "oForm." + cTempName)

* Copy properties
* Get properties for the original object
* Note work-around in the next line, using "#+" for flags, when
* only "#" is needed.
nOldMembers = AMEMBERS(aOriginalProps, oOriginalControl, 3, "#+")
FOR nMember = 1 TO nOldMembers
    DO CASE
    CASE "R" $ UPPER(aOriginalProps[ nMember, 5])
        * Read-only, so skip it
    CASE "NAME"=UPPER(aOriginalProps[ nMember, 1])
        * Name property, we'll do it later
    CASE "PROPERTY" $ UPPER(aOriginalProps[ nMember, 2])
        * See whether the property has changed
        * and the new control has this property.
        * If so, copy it.
        IF "C" $ UPPER(aOriginalProps[ nMember, 5]) ;
            AND PEMSTATUS( oControl, aOriginalProps[ nMember, 1], 5)
            cPropName = aOriginalProps[ nMember, 1]
            IF EMPTY(oOriginalControl.ReadExpression(cPropName))
                oControl.&cPropName = oOriginalControl.&cPropName
            ELSE
                oControl.WriteExpression( cPropName, ;
                    oOriginalControl.ReadExpression(cPropName) )
            ENDIF
        ENDIF
    ENDIF
ENDIF
```

```

CASE INLIST(UPPER(aOriginalProps[ nMember, 2]), "METHOD", "EVENT")
  * See whether the method has changed
  * and whether the new control has this method
  * If so, copy the contents
  IF "C" $ UPPER(aOriginalProps[ nMember, 5]) ;
    AND PEMSTATUS( oControl, aOriginalProps[ nMember, 1], 5)
    cMethod = aOriginalProps[ nMember, 1]
    cMethodCode = oOriginalControl.ReadMethod(cMethod)
    oControl.WriteMethod( cMethod, cMethodCode )
  ENDIF
OTHERWISE
  * We should never get here
ENDCASE
ENDFOR

* Remove original object
cOldName = oOriginalControl.Name
oForm.RemoveObject( oOriginalControl.Name )

* Rename new object
oControl.Name = cOldName

* Select newly added object
oForm.&cOldName..SetFocus()

RETURN

```

The method asks the custom control for the class name and class library selected. It then adds a control of that class to the form, giving it a temporary name. Then, the method gets a list of the properties and methods of the control being replaced and loops through them, copying any changed values to the new control. The idea is that we want the new control's appearance and behavior, except where we've already changed it in the control being replaced. When this process is done, the original control is removed and the new control is renamed. Finally, the new control is selected, for two reasons. First, because the replaced control was selected when the builder was called. Second and more importantly, because VFP becomes unstable when you remove the control that's currently selected in the property sheet.

To use the builder, run `BaseBuilderMain.PRG` from the Command Window once to register it. From that point on, when you right-click and choose Builder for a control, this builder will be available.

## Controlling Projects with Project Hooks

From the introduction of the Project Manager, developers have written tools to manipulate projects programmatically. In VFP 6, the task became much easier as objects were added to provide programmatic access to project contents without manipulating the underlying table. The Project and File objects, along with the corresponding collections (Projects and Files), make it easy to examine and modify the contents of a project. However, these objects are COM objects and cannot be subclassed in VFP to add event or method code.

Instead, at the same time, the ProjectHook base class was added. This is a native VFP class with events that fire when things happen to the associated project. For example, the ProjectHook class has a BeforeBuild event that fires when the process of building a project into an APP, EXE or

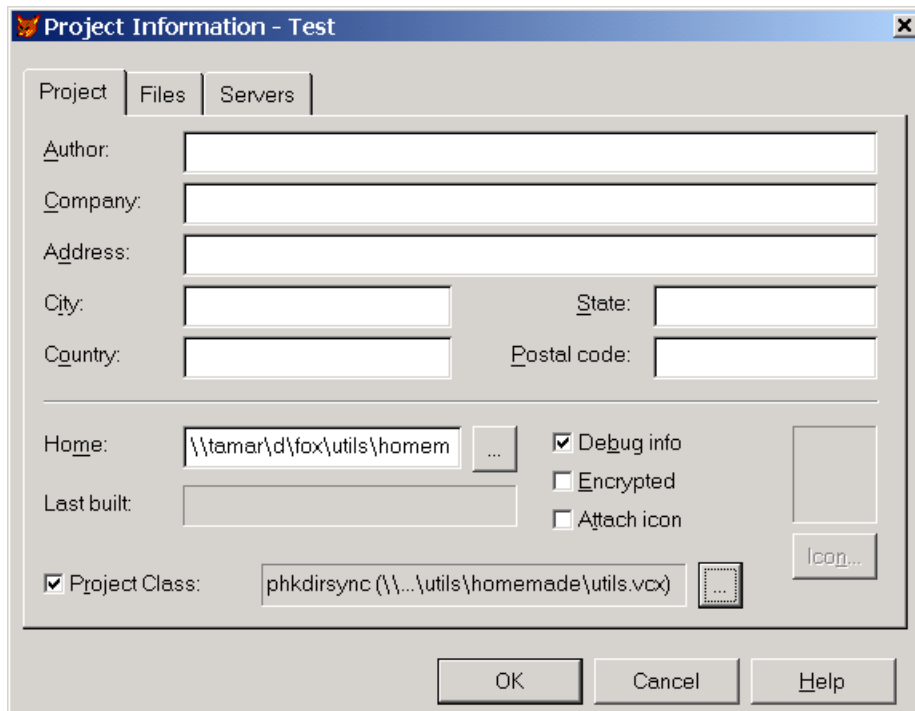
DLL begins. The AfterBuild event fires when the build is completed (whether successful or not). A collection of QueryXXXFile events fire when a file is added, changed, deleted or run. So, by subclassing the ProjectHook class and adding code, it's possible to have actions occur as a developer works on a project.

In VFP 7, the ProjectHook class exposes three new events: Activate, Deactivate and QueryNewFile. Activate and Deactivate fire when the project receives and loses focus, respectively. QueryNewFile fires at the beginning of the creation of a new file for the project (for example, when the developer clicks the New button).

Project hooks can do many different things. The prototypical example is "poor man's source control," in which every change to the project is logged. Many developers use project hooks to perform an assortment of actions at build time – for example, to remove printer-specific information from reports.

## Specifying a Project Hook

A project hook can be attached to a single project or shared by many. To attach a project hook to a particular project, open the project and choose Project Info from the Project menu or the context menu. In the Project Information dialog (see figure 9), check the Project Class checkbox. In the dialog that appears, navigate to the project hook class and choose it.





**Figure 9 Connecting a project hook to a project – Check the Project Class checkbox and navigate to the desired class. This dialog is accessed from the Project Info item on the Project menu and on the context menu for projects.**

When a project hook is attached in this way, it's instantiated each time the project is opened. (When you first attach the project hook, you need to close the project and reopen it to instantiate the project hook the first time.)

It's also possible to attach a project hook manually. Each project object has a ProjectHook property – to specify a project hook on the fly, instantiate it, then set the ProjectHook property of the project to the object reference:

```
oHook = CreateObject("phkMyHook")
_VFP.ActiveProject.ProjectHook = oHook
```

When you attach a project hook this way, it applies only to the current use of the project. When you close the project, the connection is lost. This is a useful approach for a special purpose project hook rather than one you want to use on an ongoing basis. It is possible for many projects to reference a single project hook object, so this technique might be used for some kind of auditing process that deals with all open projects.

## The Project object

For a project hook to be really useful, it needs to manipulate the Project object and its contents. So, it's useful to know something about the structure of those objects.

The \_VFP reference to the FoxPro application server has a Projects collection containing one item for each open project. \_VFP.Projects.Count property indicates the number of open projects. The ActiveProject property of \_VFP (used in the example above) contains an object reference to the current project.

Each Project object has a number of properties, such as HomeDir (for the home directory of the project), Name (the path and filename for the project), and BuildDateTime (the date and time of the most recent build). It also has a Files collection, containing one item for each file in the project. (Note that items consisting of two files, like a form or menu, have a single entry in the Files collection.) Like Project, Files has a Count property that indicates how many items are in the collection.

The File object has properties like Name (the path and file name), Description (the description for this file in the project), LastModified (the date and time of the last modification to the file), and Exclude (indicates whether the file should be excluded when building the project).

Both Project and File have a variety of methods. For File, the methods you're most likely to use are Modify, Remove and Run, which do exactly what their names suggest. The remaining File methods relate to integration with source control.

The Project object has only five methods: Build (to build the project), CleanUp (packs the project file), Close (to close the project), Refresh (updates the display of the project), and SetMain (sets a file as the main file for the project).

## Creating a Project Hook

To create a project hook, you subclass the ProjectHook base class and add code to events. For any project hook that's intended to work with a single project at a time (that is, to be attached through the Project Info dialog), it's a good idea to set up a reference to the object.

Add an oProject property to the class. Then, in the Init method, put this code:

```
This.oProject = _VFP.ActiveProject
```

At the time the project hook is instantiated, the project it's attached to is the active project. Later on, when project hook code runs, that may not be the case.

It's also a good idea to clean up in the Destroy method:

```
This.oProject = .null.
```

## An example: Returning to the project directory

With the addition of the Activate method in VFP 7, it's possible for a project hook to easily restore the project environment when focus returns to the project. One such environmental setting is the current directory. When working on a project, it can be useful to have its home directory set as the current directory. A project hook to accomplish this task (phkDirSync in Utils.VCX in the session materials) needs only a few lines of code, in the Activate method:

```
* Move to the home directory of the project
DODEFAULT()
IF VARTYPE(This.oproject) = "O"
    CD (This.oproject.HomeDir)
ENDIF
```

Unfortunately, the Activate method doesn't fire in a couple of situations where you might expect it to. When a project opens because VFP is set to reopen the last project at start-up time (an option on the General page of the Options dialog), the Activate method doesn't fire. The work-around for this case is simple: add the same code to the Init method of the project hook. Init does fire on the way in.

The other problem time is when the project is docked. In that case, clicking on it doesn't fire the Activate method. That's probably because toolbars don't get focus and the project can be seen as a toolbar when it's docked. Unfortunately, understanding the problem doesn't solve it and I have yet to find a work-around for this case.

## Hooking Project Hooks

Since different projects may call for different operations, writing a number of project hooks that each perform a single task seems a much better strategy than creating a single monster project hook that does everything conceivable. (Individual single-purpose project hooks are also easier to debug and maintain than "the mother of all project hooks.") However, a project can have only one project hook attached at a time.

One solution to this problem is to manually attach each project hook when you need it, but this approach takes away a lot of the utility of project hooks. You have to know what you need and when.

A better approach is to hook project hooks together, creating a chain of hooks that can all be attached at the same time, with each handling its own operations. When an event fires, the first hook in the chain passes it to the next, which passes it to the next, and so on down the line until the last project hook is reached. Each, in turn, performs whatever operation it has for that event.

The easiest way to create chained project hooks is to start with a project hook subclass that facilitates chaining. Then, subclass all of your project hooks from that class. The session materials include `phkBase`, a first-level project hook subclass that provides chaining.

`phkBase` has a custom `oHook` property used to attach one project hook to the next in line. `oHook` has an `Assign` method that ensures the chain is maintained when the `oHook` property is set:

```
LPARAMETERS vNewVal

IF NOT ISNULL( m.vNewVal)
  * Adding to the chain
  IF VARTYPE( m.vNewVal ) = "O" ;
    AND PEMSTATUS( m.vNewVal, "oHook", 5)
      m.vNewVal.oHook = This.oHook
    ENDIF
  ENDIF
ENDIF

THIS.oHook = m.vNewVal

RETURN
```

Each event method passes the call down the chain and checks the results. Here's the code for `QueryModifyFile`, for example:

```
LPARAMETERS oFile, cClassName
* Pass the call along to the other project hooks in the chain.

LOCAL cMethod, cParams, lReturn

cMethod = JUSTEXT( PROGRAM())
IF NOT EMPTY( cClassName )
  cParams = "'" + cClassName + "'"
ENDIF

lReturn = This.PassMethodCall( cMethod, cParams, oFile )

IF NOT lReturn
  NODEFAULT
ENDIF

RETURN lReturn
```

The `PassMethodCall` method is a centralized facility for passing calls down a level.

```
* Pass a method call up the calling chain.
LPARAMETERS cMethod, cParams, oFile

ASSERT VARTYPE(cMethod) = "C" AND NOT EMPTY(cMethod);
  MESSAGE "phkBase.PassMethodCall: Must pass method name"
```

```

ASSERT VARTYPE(cParams) = "C" OR ;
    (VARTYPE(cParams) = "L" AND NOT cParams) ;
    MESSAGE "phkBase.PassMethodCalls: Pass cParams as a string or omit"

ASSERT VARTYPE( oFile ) = "O" OR (VARTYPE(oFile) = "L" AND NOT oFile) ;
    MESSAGE ;
    "phkBase.PassMethodCalls: Pass oFile as object reference or omit"

LOCAL cCall, lReturn

IF VARTYPE(This.oHook) = "O" ;
    AND PEMSTATUS( This.oHook, cMethod, 5)
    cCall = "This.oHook." + cMethod + "("
    IF VARTYPE(oFile) = "O"
        cCall = cCall + "oFile"
    ENDIF
    IF VARTYPE(cParams) = "C"
        IF VARTYPE(oFile) = "O"
            cCall = cCall + ", "
        ENDIF
        cCall = cCall + cParams
    ENDIF
    cCall = cCall + ")"
    lReturn = &cCall
ELSE
    lReturn = .T.
ENDIF

RETURN lReturn

```

The phkBase class also has an abstract AttachHook method. Its purpose is to set up the chain of hooks. One good approach is to create all your special-purpose hooks as subclasses of phkBase. Then, for a given configuration of project hooks, create a subclass that contains code only in the AttachHook method – that code sets up the chain of hooks.

Thanks to the oHook\_Assign code, all you need to do is set the oHook property to each project hook in turn. Behind the scenes, they all get hooked together. The following code in the AttachHook method chains the current project hook with the phkDirSynch and phkForceDescription project hooks in the Utils class library in the session materials. (Due to a bug in the release version of VFP 7, the phkForceDescription project hook doesn't work correctly.)

```

This.oHook = NEWOBJECT("phkDirSynch", "Utils")
This.oHook = NEWOBJECT("phkForceDescription", "Utils")

```

Additional assignments can be added to the list if other project hooks are to be used.

## Borrow, Don't Build

When you have a special need for extending a tool, building your own may be the only way to go. However, plenty of other people work with VFP and there are a number of add-ins, builders, project hooks and other extensions available. Check out the libraries of the CompuServe VFP forum ([go.compuserve.com/msdevapps](http://go.compuserve.com/msdevapps)), the Universal Thread ([www.universalthread.com](http://www.universalthread.com)) and the Virtual FoxPro User Group ([www.vfug.org](http://www.vfug.org)), as well as the FoxPro Wiki ([fox.wikis.com](http://fox.wikis.com)), for

ideas and downloadable extensions. You may also want to check back issues of the two major FoxPro publications, FoxTalk and FoxPro Advisor, for add-ins, builders, and other extensions.

A few people have been kind enough to let me include their tool extensions with the materials for this session. You'll find them in the AddlTools directory in the code for this session.

## Resources

The documentation provided by Microsoft for extending the tools varies in both quantity and quality. For some of the tools, there are a number of articles and white papers. A few have been covered extensively in books. For other tools (especially the newer ones), very little is available.

More information can be found about FoxPro Advisor at [www.advisor.com](http://www.advisor.com). Information about FoxTalk is at [www.pinpub.com/ft](http://www.pinpub.com/ft).

## Builders

Granor and Roche, *Hacker's Guide to Visual FoxPro 6.0*, Hentzenwerke Publishing ([www.hentzenwerke.com](http://www.hentzenwerke.com)), Section 5.

Hennig, *Building and Using VFP Developer Tools*, [www.stonefield.com/techpap.html](http://www.stonefield.com/techpap.html). This zip file you download for this paper includes several builders and some project hooks.

Hennig and Morhart, *Reusable Tools: Building Your Own Builders with BuilderB*, FoxTalk, 12/97

Hennig, *Reusable Tools: Building Builders with BuilderD*, FoxTalk, 3/99

Hennig, *Reusable Tools: Modifying VFP's Wizards and Builders*, FoxTalk, 8/99

## Class Browser

Drochak and Popp, *Improving VFP 6.0's Class Browser*, FoxTalk, 6/99.

Egger, *Advanced Object Oriented Programming with Visual FoxPro 6.0*, Hentzenwerke Publishing ([www.hentzenwerke.com](http://www.hentzenwerke.com)), Chapter 5.

Granor, *Have it Your Way!*, FoxPro Advisor, 4/01.

Granor and Roche, *Hacker's Guide to Visual FoxPro 6.0*, Hentzenwerke Publishing ([www.hentzenwerke.com](http://www.hentzenwerke.com)), Section 5.

Hodder, *Tracking Classes and Libraries Modified By the Class Browser*, [http://www.inquiry.com/techtips/vfox\\_pro/10min/10min1000/10min1000.asp](http://www.inquiry.com/techtips/vfox_pro/10min/10min1000/10min1000.asp)

Levy, *Extend the Visual FoxPro Class Browser with Add-Ins*, FoxTalk, 1/96.

Liskin, *Using the Class Browser*, FoxPro Advisor, 10/99

MSDN, *The Visual FoxPro 6.0 Class Browser*, <http://msdn.microsoft.com/vfoxpro/technical/articles/oop.asp>.

## Component Gallery

Egger, *The Component Gallery: VFP's Best Kept Secret*, FoxTalk, 4/01.

Egger, *Advanced Object Oriented Programming with Visual FoxPro 6.0*, Hentzenwerke Publishing ([www.hentzenwerke.com](http://www.hentzenwerke.com)), Chapter 5.

Granor and Roche, *Hacker's Guide to Visual FoxPro 6.0*, Hentzenwerke Publishing ([www.hentzenwerke.com](http://www.hentzenwerke.com)), Section 5.

## Coverage Profiler

Nicholls, *Visual FoxPro Coverage Profiler Add-ins and Subclasses*, [http://spacefold.com/lisa/LSN\\_CoverageExtend.htm](http://spacefold.com/lisa/LSN_CoverageExtend.htm)

Rubel, *Coverage and Profiling*, FoxPro Advisor, 12/98.

Rubel, *Extend the Coverage Application*, FoxPro Advisor, 1/99

## Project Hooks

Hennig, *Building and Using VFP Developer Tools*, [www.stonefield.com/techpap.html](http://www.stonefield.com/techpap.html). This zip file you download for this paper includes several builders and some project hooks.

Hennig, *The Happy Project Hooker*, FoxTalk, 9/98

Liskin, *Hooking into the Project*, FoxPro Advisor, 5/99

## Acknowledgements

Jim Booth contributed to both the design and the code for the Class Browser add-in described here. Doug Hennig tested several of the tools here and inspired several of the code samples. Lisa Slater Nicholls was kind enough to make her article on the Coverage Profiler accessible on-line. Thanks, also, to all the people who provided the tool extensions included with the session materials.

*Copyright 2001, Tamar E. Granor, Ph.D.*