Make Thor Your Own

Tamar E. Granor
Tomorrow's Solutions, LLC
www.tomorrowssolutionsllc.com
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

While using Thor exactly as it's installed offers lots of benefits, Thor also can be customized in lots of ways. From adding your own tools to specifying plug-ins that change the behavior of tools to setting up options for tools, you can tailor Thor's behavior to provide yourself and your team exactly what you need. The Thor Framework and Thor Procs provide basic code that makes it easier to customize Thor.

In this session, we'll see how to take Thor to the next level by adding tools, setting up options, and modifying plug-ins. We'll also explore the Thor Framework and Thor Procs to understand the rich environment Thor provides.

Thor comes with an incredible collection of tools that make developing software with VFP easier. If that were all it offered, it would be worth installing. However, Thor is also a container for any tools you want to add. It also offers a variety of ways to customize the behavior of the tools it manages.

Thor Architecture

You don't need to know the details of Thor's internals in order to customize it. But a basic understanding of how it works will make the rest of this paper easier to follow.

One of the principle goals for Thor was that it would survive a CLOSE ALL, CLEAR ALL sequence, which most Xbase tools do not. To facilitate that, when you run Thor, it adds a property to _SCREEN called cThorDispatcher, and stores a bundle of VFP code there. When you choose an action in Thor, it runs by executing the code that's stored in _SCREEN.cThorDispatcher, passing parameters to indicate the desired action. The ExecScript() function makes this possible; **Listing 1** shows the basic structure.

Listing 1. This is the typical way to execute any code inside Thor.

```
ExecScript(_Screen.cThorDispatcher, "Instruction", "Params")
```

The first parameter to ExecScript() is the code to execute, in this case, the code stored in _SCREEN.cThorDispatcher. Subsequent parameters to ExecScript() are passed to the executing code. The Thor convention is that the second parameter to ExecScript() is an instruction, the action you want to take. The third and later parameters indicate what you're operating on.

You'll see code like **Listing 1** throughout this paper.

Adding tools

I have a number of small tools I've written at one time or another, mostly with no user interface. For example, one of them goes through a project's folders and fills a cursor with the names of items in the folders that aren't included in the project. The whole thing is about 50 lines of code.

The problem with little tools like this is that when I want to use them, I have to find them and look at the code to remember how they work. Then I have to either make sure the code is in the path or specify the full path in order to run the tool.

One of the design criteria for Thor was to make it easy for people to add tools and to share them. That way, you can take all the little tools like the one described above and stick them into the Thor Tools menu to keep them handy. Among other benefits, Thor manages the code so you don't have to worry about paths.

To add a tool to Thor, open the Thor configuration form (Thor | Configure from the menu) and click the Tool Definitions tab. Click the Create Tool button to open the Create Tool dialog, shown in **Figure 1**. Once you give the tool a name using the textbox preceded by

"Thor_Tool_," click the Create button to open a template file for the tool. (The string "Thor_Tool_" becomes part of the tool's filename.)

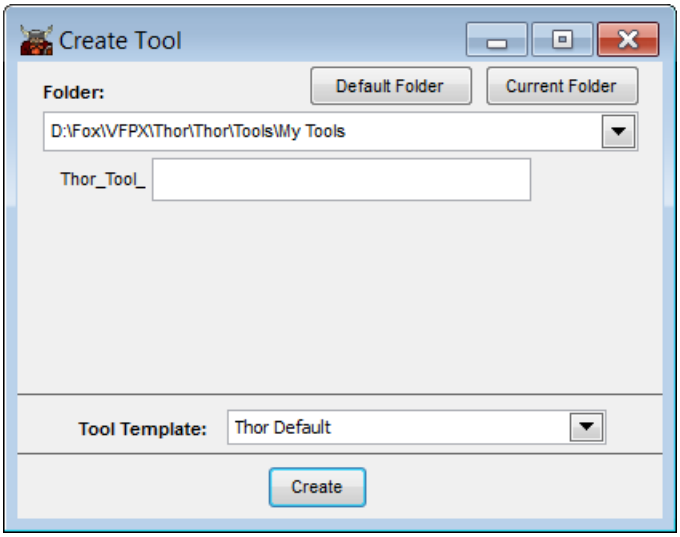

Figure 1. To add a new tool to Thor, specify the name of the tool in this dialog and click Create.

Thor tools require a specific format, which is provided by the template. The default template is shown in **Listing 2**. The bulk of the template provides a place to give Thor information about this tool; to create a tool, fill in one or more of the properties listed. The Prompt property is required and contains the prompt that will appear on the Thor Tools menu. The Description property appears only in the Thor configuration dialog and Tool Launcher. When you search in Tool Launcher, both the prompt and the description are searched (as are the Category and Author properties).

Listing 2. The default Thor template shows exactly what's required to create a Thor tool.

```
Lparameters lxParam1

*************************************************************
*************************************************************
* Standard prefix for all tools for Thor, allowing this tool to
*   tell Thor about itself.

If Pcount() = 1                     ;
      And 'O' = Vartype (lxParam1)  ;
      And 'thorinfo' == Lower (lxParam1.Class)

   With lxParam1

      * Required
      .Prompt         = 'Prompt for the tool' && used in menus

      * Optional
      Text to .Description NoShow && a description for the tool
Enter a description for the tool here
      EndText
      .StatusBarText = ''
```

```
      .CanRunAtStartUp = .T.

      * These are used to group and sort tools when they are displayed in menus or
      * the Thor form
      .Source         = '' && where did this tool come from?
                           && Your own initials, for instance
      .Category      = '' && creates categorization of tools;
                           && defaults to .Source if empty
      .Sort         = 0 && the sort order for all items from the same Category

      * For public tools, such as PEM Editor, etc.
      .Version      = '' && e.g., 'Version 7, May 18, 2011'
      .Author        = ''
      .Link          = '' && link to a page for this tool
      .VideoLink     = '' && link to a video for this tool

   Endwith

   Return lxParam1
Endif

If Pcount() = 0
   Do ToolCode
Else
   Do ToolCode With lxParam1
Endif

Return

************************************************************
************************************************************
* Normal processing for this tool begins here.
Procedure ToolCode
   Lparameters lxParam1

EndProc
```

The Category and Sort properties let you specify where the tool appears in the list of Thor tools. That list appears in the Thor Tools menu, in the Configuration form and in the Launcher. If Category is specified, the tool appears in that group; you can specify multiple levels in the menu by separating the items with the vertical bar ("|"). For example, to add an item to the Misc. group in the Code menu, specify "Code|Misc."

You might use your initials or your company name to group all of your own tools together.

The Source property, which can also be used for this purpose, is deprecated, so just leave that property empty.

The Sort property determines the position of this item in the specified submenu.

The last set of properties in the template is relevant only for tools being shared with the VFP community.

Listing 3 shows the properties set for my tool that lists files unused in a project. (The trailing comments for the properties have been removed to save space, as have some properties I'm not setting.)

Listing 3. The Thor properties set for the Unused files tool.

```
    * Required
    .Prompt       = Unused files'

    * Optional
    Text to .Description NoShow
List files in project folders that aren't used in project.
    EndText
    .StatusBarText = ''

    * These are used to group and sort tools
    * when they are displayed in menus
    * or the Thor form
    .Source       = 'TSLLC'
    .Category      = ''
    .Sort       = 0
```

The heart of the tool is the ToolCode procedure; that's where you put the code to perform the task. **Listing 4** shows the code added to ToolCode for the Unused files tool. As you can see, it's not terribly complex. It checks for an active project, and if one is found, the list of files is copied into an array. The current path is then parsed and stored to provide a list of folders to check. The main loop processes those folders one at a time. For each, ADIR() returns the list of files in the folder. For each file that could be in the project, based on its extension, the array of project files is checked. If the file isn't in the array, it's added to the list of unused files.

Listing 4. The ToolCode procedure for the Unused files tool.

```
LOCAL oProject, nCounter, oFile
LOCAL nDirs, nDir, aDirs[1]

IF _vfp.Projects.Count = 0
   MESSAGEBOX("You must open a project before using this tool.")
   RETURN
ENDIF

oProject = _VFP.ActiveProject
* First, make a list of all files in project
LOCAL aProjFiles[oProject.Files.Count]

nCounter = 0
FOR EACH oFile IN oProject.Files
   nCounter = m.nCounter + 1
   aProjFiles[m.nCounter] = UPPER(oFile.Name)
ENDFOR

CREATE CURSOR Unused (mFileName M)
```

```
* Now traverse directories
* First, make a list of directories
* using current path
nDirs = ALINES(aDirs, SET("Path"), 1, ";", ",")
CREATE CURSOR DirsToCheck (mDirName M)

* Start with project's home folder.
INSERT INTO DirsToCheck VALUES (oProject.HomeDir)

FOR nDir = 1 TO m.nDirs
   INSERT INTO DirsToCheck VALUES (FULLPATH(aDirs[m.nDir], oProject.HomeDir + "\"))
ENDFOR

LOCAL aFiles[1], cOldDir, cFile, nFilesToCheck, cExt

cOldDir = SET("Default") + CURDIR()

SCAN
   IF DIRECTORY(mDirName)
      CD ALLTRIM(mDirName)
      nFilesToCheck = ADIR(aFiles, "*.*")
      FOR nFile = 1 TO m.nFilesToCheck
         cFile = aFiles[m.nFile, 1]
         cExt = JUSTEXT(m.cFile)
         IF INLIST(cExt, "PRG", "SCX", "MNX", "FRX", "VCX", "QPR")
            IF ASCAN(aProjFiles, FORCEPATH(m.cFile, ALLTRIM(mDirName)), ;
                     -1, -1, 1, 7) = 0
               INSERT INTO Unused VALUES ;
                  (FORCEPATH(m.cFile, ALLTRIM(DirsToCheck.mDirName)))
            ENDIF
         ENDIF
      ENDFOR
   ENDIF
ENDSCAN

USE IN SELECT("DirsToCheck")

CD (m.cOldDir)

RETURN
```

Once you've specified the necessary properties and added code to the ToolCode procedure, save the program. It automatically gets saved in the right place with the right name.

To test your tool, either close the Thor configuration form, or click its Thor button. Either one refreshes the menu. Once you do so, the new tool is included in the Thor Tools menu, as in **Figure 2**. In the other Thor forms that list tools (the Configuration form and the Launcher), your added tool shows up with a yellow background.

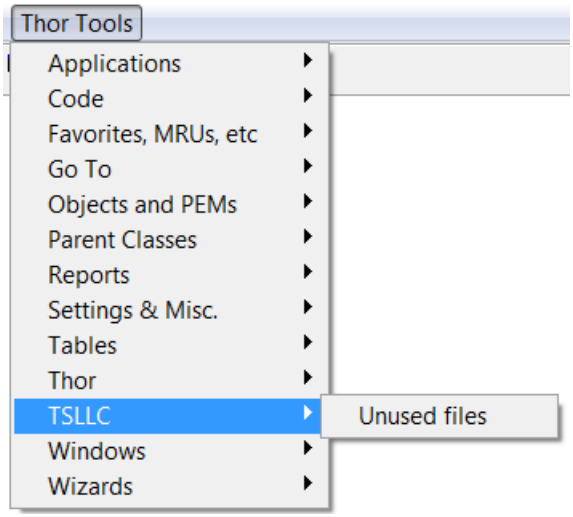

Figure 2. Once you finish the definition of a new tool and refresh Thor, the tool is shown on the menu.

To give the Unused files tool a try, simply choose it from the Thor menu. If you don't have an open project, you'll see a dialog to that effect. Open a project (and SET PATH to the folders for that project) and try again. When the tool finishes, you'll find a cursor named Unused, with one record for each unused file.

This version of the Unused files tool is included in the session materials as Thor_Tool_Unused_Files_Take_1.PRG.

Editing your Thor tools

Once you've created a Thor tool, you can edit it through Thor, just as you can edit the built-in tools.

To open the code for an existing tool, start in either the Launcher or the Configuration form. In either case, highlight the tool you're interested in (in the Configuration form, on the Tool Definitions page) and click the Edit Tool button at the bottom of the right pane. When you open a tool's code this way, it's added to the MRU ("most recently used") list for PRGs, so you can easily reopen it right from the Command Window.

For your custom tools, the code opens right up. For built-in Thor tools, it takes a little more work.

The Edit File dialog, shown in **Figure 3**, appears. If you just want to look at the code, click View this file in Read-Only mode. If you want to customize the tool, click Copy this file to folder 'My Tools' and edit it. In this case, a copy of the tool code opens; when you save it, it's stored in the My Tools folder of your Thor installation, and from that point on, when you use the tool, the copy in My Tools is used. (Note that the copy isn't saved until you actively save it, so if you start working on modifying a tool and discard your changes, the original tool stays in effect.) This allows you to make changes while retaining the original, and means that if the tool is updated in the Thor Repository, when you update Thor, your customized copy remains intact (though, of course, it won't reflect the updates from the

Repository—see “The Thor Framework” later in this document for a way to extend rather than replace tool code).

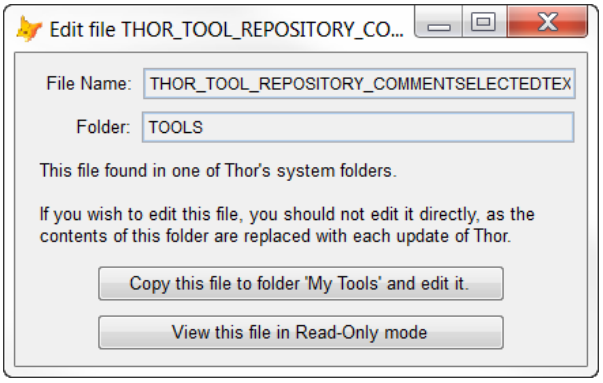

Figure 3. This dialog appears when you click the Edit Tool button in either the Launcher or the Configuration form.

Adding Options to Thor Tools

Given the dozens of tools that come with Thor, it’s no surprise that there are a variety of opinions about how some tools should operate.

For example, one of the tools that comes with Thor is Comment Highlighted Text. As the name suggests, it turns whatever lines are currently highlighted into comment lines. It also adds a comment before those lines. By default, that comment says “Removed” and the date. But it’s easy to see that different developers might want different versions of that comment.

Thor allows you to set the comment to use. The Options tab of the Thor Configuration form (shown in **Figure 4**) lets you specify a string to use. (As the figure shows, the string is run through textmerge first.)

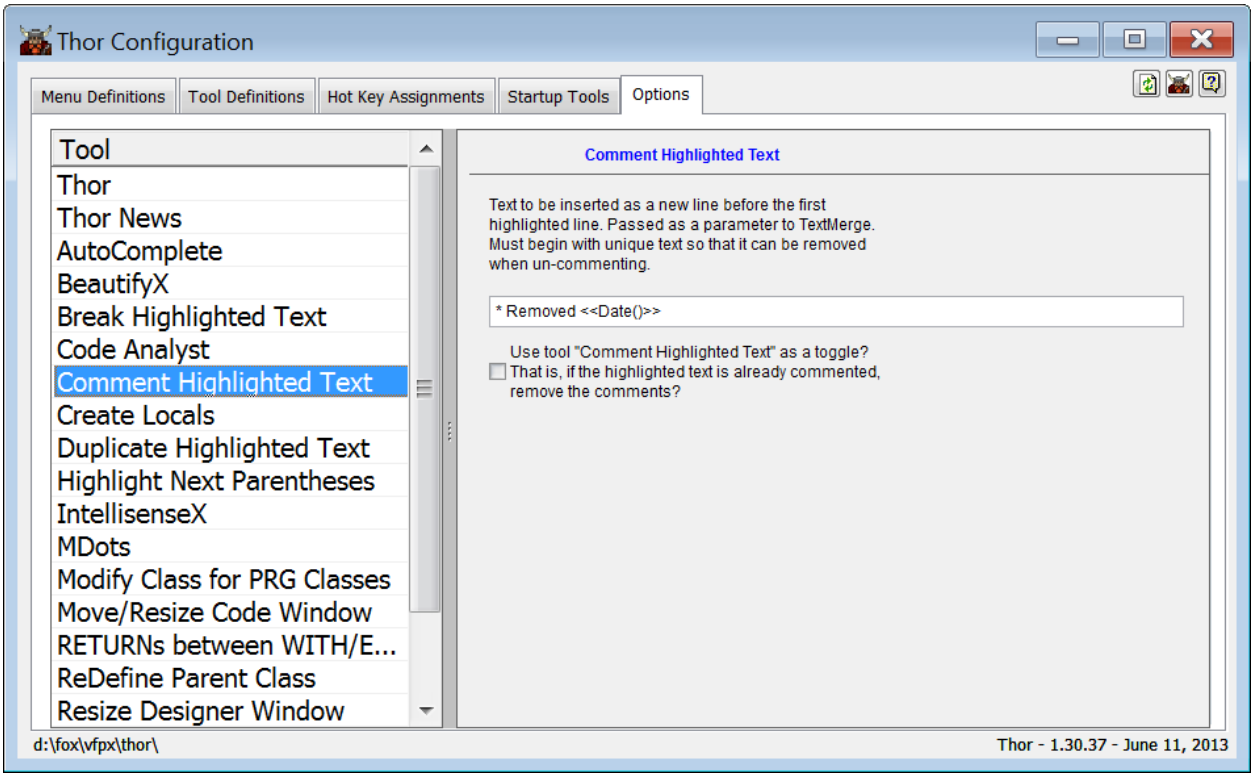

Figure 4. The Options tab of the Thor Configuration dialog lets you specify the string to use as a header comment when you use the Comment Highlighted Text tool.

So, if you prefer the header comment for removed code to, say, include your initials, you can just add them to the string. **Figure 5** shows the Options tab after I modified the string to include my initials and some additional text.

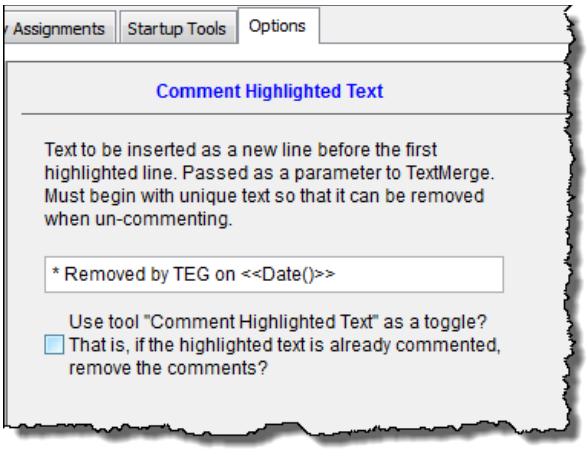

Figure 5. Modifying the header comment on the Options tab changes what the Comment Highlighted Text tool inserts.

The architecture for specifying options for Thor Tools is quite elegant and makes it easy to add options to any tool. There are three elements involved: defining an option, displaying an option and accessing an option.

In most cases, all the elements for specifying options appear in the tool code. You can follow along with this example by opening the code for the Comment Highlighted Text tool (in read-only mode).

Defining options

The first step in adding options is to figure out what options you want to offer. Once you have a list, there are two steps to telling the tool itself about them.

Options are specified by creating a class definition for each option and then telling the tool about those classes.

For each option you want, you need to add a subclass of Custom to the tool's PRG file. Each option class needs four properties, set as follows:

- Tool is the name that appears in the left pane of the Options page.
- Key is a unique name (within the tool) for this option. It identifies this option.
- Value is the initial value for the option. It can be character, numeric, logical or date. If no initial value is specified, the option is null until the user specifies a value.
- EditClassName is the name of a class that displays the option. There are two ways to do so. The first is to name a container class defined in the same PRG that contains the instructions for displaying the options for this tool. The alternative is to use a string in the form "classname of filename.vcx", pointing to an existing VFP class that can be dropped onto the Options page. The class library should be stored in Thor's My Tools folder. (In either case, a single option container can include options for multiple tools.)

The value of EditClassName should be the same for all options for a single tool.

Listing 5 shows the option class definitions for the Comment Highlighted Text tool. The constants they reference appear at the top of the code for the tool and are shown in **Listing 6**.

Listing 5. Create a custom class to define each option for a tool.

```
Define Class clsAddComments As Custom

  Tool          = ccXToolName
  Key           = ccCommentText
  Value         = '* Removed <<Date()>>'
  EditClassName = ccContainerClassName

Enddefine
```

```
Define Class clsToggleComments As Custom

  Tool          = ccXToolName
  Key           = ccToggleComments
  Value         = .F.
  EditClassName = ccContainerClassName
```

Listing 6. The tools that come with Thor use constants to make it easier to manage the tool names and option keys.

```
#Define    ccContainerClassName  'clsCommentSelectedText'
#Define    ccXToolName           'Comment Highlighted Text'

#Define    ccCommentText         'Comment Highlighted Text'
#Define    ccToggleComments      'Toggle Comments'
```

Listing 7 shows the alternative approach to specifying the display of options, using an existing visual class. This line is drawn from one of the various tools that make up IntelliSenseX.

Listing 7. You can specify the controls that let users set options using VFP classes.

```
 EditClassName = 'clsISX of Thor_Options_ISX.VCX'
```

Once you've created the classes for each option, you need to tell the tool about them. In the top portion of the tool definition, add two properties, OptionClasses and OptionTool. OptionClasses is a comma-separated list of names for classes that define the individual options. **Listing 8** shows the two properties as they're specified for the Comment Highlighted Text tool.

OptionTool indicates what tool is selected on the Thor Options page when the user clicks the Options button for the tool (in the Thor Configuration form or the Launcher); the Options button only appears if a value has been assigned to this property. The value in this property appears in the left pane of the Options page. (Note that this structure means that multiple tools can share a set of options. For example, all the individual tools that comprise IntelliSenseX have a single set of options labeled "IntelliSenseX.")

Listing 8. To specify options for a tool, add two properties to the tool definition.

```
.OptionClasses = 'clsAddComments, clsToggleComments'
.OptionTool    = 'Comment Highlighted Text'
```

Displaying options

The next step in providing options is indicating how to display them. All options are shown on the Options tab of the Thor Configuration form (**Figure 6**). The left pane shows the list of tools for which options are available. Choose a tool in that list and its options appear in

the right pane. Thor and Thor News are always listed first, then other tools in alphabetical order.

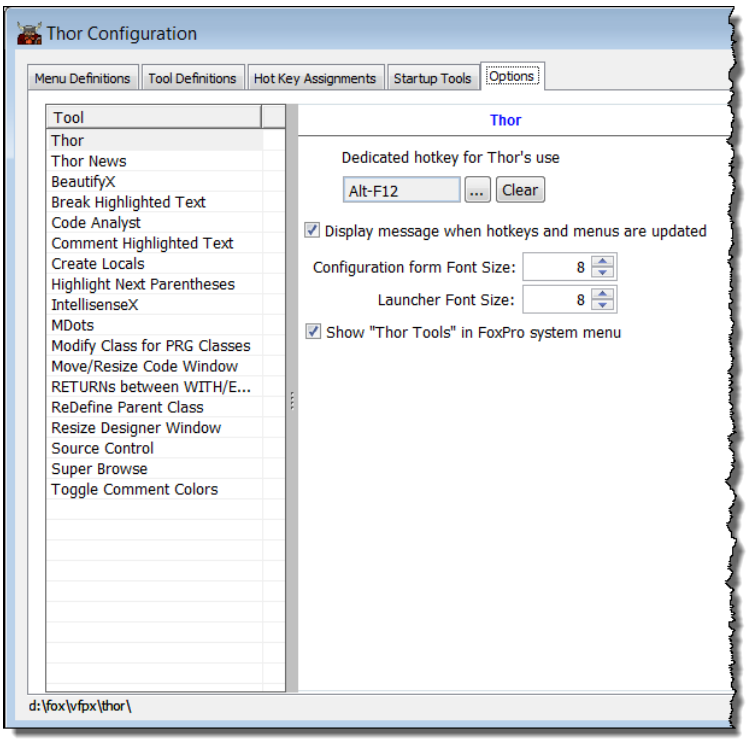

Figure 6. The Options tab of the Thor Configuration form shows the options for each tool that has them. Choose the tool in the left pane to see its options in the right pane.

As indicated in the previous section, there are two ways to specify the user interface for setting options. Options may use the VFPX Dynamic Forms project to specify layout. Alternatively, they can use standard VFP classes.

Displaying options with Dynamic Forms

Dynamic Forms lets you specify form layout using a mark-up syntax that looks something like property assignments. (A complete explanation of Dynamic Forms is beyond the scope of this paper, but the Dynamic Forms page of VFPX points to a couple of videos and links to documentation.) Fortunately, you can create option pages for tools without knowing too much of it; just model your code after the code used for existing tools.

To specify an options page, you add a definition for the class that was referenced in the EditClassName property of the individual option items. That class needs code only in its Init method. In that code, you instantiate a class called OptionRenderEngine that's built into Thor. Then you set the OptionRenderEngine's cBodyMarkup property to the Dynamic Forms mark-up needed for your options. Finally, you call the OptionRenderEngine's Render method.

Listing 9 shows the clsCommentSelectedText class that's part of the Comment Highlighted Text tool. Note that the values for the Caption properties wrap here, but in the actual tool code, run on a single line.

Listing 9. One way to display options for a Thor Tool uses the Dynamic Forms project.

```
Define Class clsCommentSelectedText As Container

Procedure Init
  loRenderEngine = Execscript(_Screen.cThorDispatcher, 'Class= OptionRenderEngine')

  Text To loRenderEngine.cBodyMarkup Noshow Textmerge

    .Class     = 'Label'
    .Caption   = 'Text to be inserted as a new line before the first highlighted
line. Passed as a parameter to TextMerge. Must begin with unique text so that it can
be removed when un-commenting.'
    .Width     = 300
    .Left     = 25
    .WordWrap = .T.
  |
    .Class     = 'TextBox'
    .Width     = 300
    .Left     = 25
    .cTool     = ccXToolName
    .cKey     = ccCommentText
    |
    .Class     = 'CheckBox'
    .Width     = 300
    .Left     = 25
    .WordWrap = .T.
    .Caption   = 'Use tool "Comment Highlighted Text" as a toggle? That is, if the
highlighted text is already commented, remove the comments?'
    .cTool     = ccXToolName
    .cKey     = ccToggleComments

  Endtext

  loRenderEngine.Render(This, ccXToolName)

Endproc
```

There are a few things to note in this code. The class is subclassed from the base Container class. All option display classes must be subclassed from Container or PageFrame or a subclass of Container or PageFrame, to provide an object that Thor can simply drop onto the Options page.

You separate specifications for individual controls with the vertical bar ("|"). The code in Listing 9 specifies three controls: a label, a textbox and a checkbox.

For each control, you use actual VFP properties to indicate the layout. Two additional properties are needed for those controls that map to option values. cTool specifies the tool

to which the option applies; this is the same name you specify for the OptionTool property in the top portion of the tool definition. cKey is the key for the specific option; this is the same as the Key property of the class that defines the option.

For example, in Listing 9, both the textbox and the checkbox have cTool set to 'Comment Highlighted Text'. The textbox also has cKey set to 'Comment Highlighted Text', which is the key for the option that determines the header comment, while the checkbox has cKey set to 'Toggle Comments', the key specified for the option that determines whether this tool operates as a toggle.

Instantiation of OptionRenderEngine follows the normal Thor style for instantiating classes provided with Thor. It uses a call to ExecScript() passing _Screen.cThorDispatcher as the first parameter and a description of what to do as the second. See "Thor Architecture," earlier in this paper and "The Thor Framework," later in this paper, for more on calling Thor with ExecScript().

Displaying options with VFP classes

The alternative approach to displaying options is to create a visual class that includes all the necessary controls and specify that class in EditClassName. Because it predates the Dynamic Forms approach, this approach is widely used by the tools that come with Thor. It's a good choice for more complex sets of options, where a pageframe may be the appropriate organizing mechanism.

The class needs to be derived from Container or PageFrame. At runtime, the specified class is dropped into the appropriate container in the Configuration dialog. That container has two custom methods to store and retrieve option values:

- GetOption(Key, Tool) retrieves the current value for the specified option;
- SetOption(Key, Tool, Value) sets the value for the specified option.

For example, the IntelliSenseX – by dot tool has quite a few options. They all specify 'clsISX of Thor_Options_ISX.VCX' for EditClassName, as in **Listing 10**.

Listing 10. The options for IntelliSenseX use a custom container class to specify options.

```
Define Class clsAlphaSort As Custom

   Tool        = 'IntellisenseX'
   Key         = 'Alpha Sort'
   Value       = .T.
   EditClassName = 'clsISX of Thor_Options_ISX.VCX'

Enddefine
```

Figure 7 shows clsIsx as it appears in the Class Designer. It has custom GetOption and SetOption methods that call the same-named methods of its parent (which, at runtime, is the Options page of the Configuration form). The controls call the container methods. For example, the UIEnable method of the "Filtering enabled" checkbox contains the code in

Listing 11, which looks up the current value of the 'Filtering' option and sets the checkbox accordingly. (Since the checkbox is on a page of a pageframe on the container, This.Parent.Parent.Parent is the container class itself.) The InteractiveChange method of the checkbox contains the code in **Listing 12**, which saves the new value and then updates the option group. The other controls in the container have similar code.

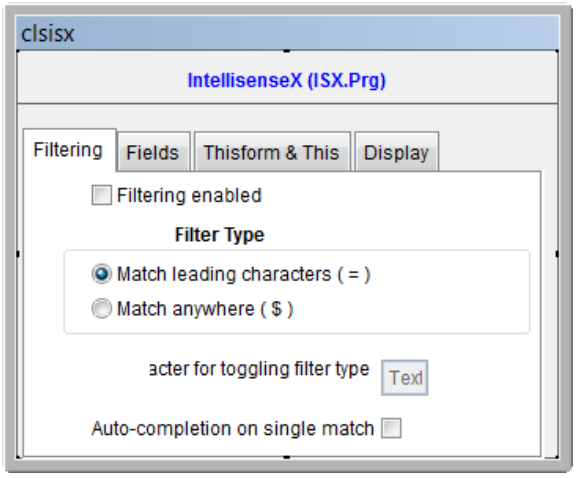

Figure 7. This class, used for specifying options for IntelliSenseX, is based on Container.

Listing 11. This code, in the UIEnable method of the "Filtering enabled" checkbox, sets the checkbox's initial value to match the stored value of the option.

```
This.Value = This.Parent.Parent.Parent.GetOption('Filtering')
```

Listing 12. The InteractiveChange method of the "Filtering enabled" checkbox saves the new setting before updating other controls.

```
This.Parent.Parent.Parent.SetOption('Filtering', This.Value)

This.Parent.opgFilterType.Refresh()
```

Accessing an option

The final part of specifying options is accessing them in tool code, so that their values affect the behavior of the tool. This part turns out to be the easiest of all, because Thor has the mechanism built right in.

To get the value of an option, you use an ExecScript() call in the form shown in **Listing 13**. You pass the key for the option (as specified in the Key property of the class that defines the option) and the tool name (as specified in the Tool property of the option class) to indicate which option you want to retrieve.

Listing 13. To retrieve the current value of an option, pass the appropriate parameters to an ExecScript() call to _Screen.cThorDispatcher.

```
uOptionValue = EXECSCRIPT(_Screen.cThorDispatcher, "Get Option =", ;
                          <Option key>, <Tool name>)
```

Listing 14 and **Listing 15** show code from the Comment Highlighted Text tool that retrieves the two options for that tool and applies them. Note that the Comment Highlighted Text option is character, while the Toggle Comments option is logical. Thor handles the different data types transparently.

Listing 14. This code retrieves the header comment to use for the Comment Highlighted Text tool, and applies textmerge to get the exact string to insert.

```
lcNewLineText = Textmerge(ExecScript(_Screen.cThorDispatcher, "Get Option=", ;
                          'Comment Highlighted text', 'Comment Highlighted text'))
```

Listing 15. This code retrieves the value of the Toggle Comments option and then applies it to determine whether to comment or uncomment the highlighted text.

```
If ExecScript(_Screen.cThorDispatcher, "Get Option=", ;
              'Toggle Comments', 'Comment Highlighted text')    ;
   And Left(Ltrim(lcClipText,' ', chr[9]), Len(lcCommentString)) == lcCommentString
  loCommentText.RemoveComments(lcClipText, ;
                               lcNewLineText)
Else
  loCommentText.AddComments(lcClipText, ;
                            lcNewLineText)
Endif
```

Using invisible options for persistent storage

The Thor options mechanism can also be used to track values behind the scenes, without providing any user interface. The Thor "Get Option=" call has a corresponding "Set Option=" call that allows you to store a value for future reference.

For example, Thor News stores the date it was last displayed, using the code in **Listing 16**. The code in **Listing 17** retrieves the stored date to determine whether it's time to show it.

Listing 16. This code in the Thor News tool stores the date that Thor News was last displayed.

```
ExecScript(_Screen.cThorDispatcher, "Set Option=", ;
           ccDateLastSeen, ccTool, Date())
```

Listing 17. This code in the Thor News tool retrieves the date Thor News was last displayed.

```
ldDataLastSeen = ExecScript(_Screen.cThorDispatcher, ;
   "Get Option=", ccDateLastSeen, ccTool)
```

This mechanism means that Thor tools can save information between runs without the tool's author having to come up with a way to do so.

Note that you can create invisible options on the fly by simply issuing a "Set Option=" call; when the specified item doesn't exist, it's created. (Of course, this means that a typo in such a call can create a new option instead of updating an existing option.)

Issuing a "Get Option=" call for a non-existent option returns .null.

Adding options to the Unused files tool

There are a couple of choices that would be useful for the Unused files tool. My original (pre-Thor) version accepts a parameter to determine whether to look in the project's home folder. In addition, it accepts a parameter listing the folders to check. Since passing such a parameter to a Thor tool is tricky, when I created the Thor version, I rewrote the code to look in the current path. (For the record, the way to pass parameters to a Thor tool is to create a Thor Proc—see "Thor Procs," later in this paper—to receive and use the parameter, and store the Proc in the My Tools folder. Then, you can create a tool to call that Proc.)

But you might prefer to look in the folders that are referenced in the project. So let's see how to add these options, and how to modify the tool code to respect the user's choices.

First, I added definitions for two classes, clsIncludeHome and clsChooseFolders, shown in **Listing 18**. Both classes specify "Unused Files" as the tool and both indicate that the class to specify them is clsUnusedFilesOptions.

Listing 18. Each option is defined by creating a class for it, based on Custom.

```
DEFINE CLASS clsIncludeHome AS Custom

Tool = 'Unused Files'
Key = 'IncludeHome'
Value = .T.
EditClassName = 'clsUnusedFilesOptions'

ENDDEFINE

DEFINE CLASS clsChooseFolders as Custom

Tool = 'Unused Files'
Key = 'ChooseFolders'
Value = 'Current path'
EditClassName = 'clsUnusedFilesOptions'

ENDDEFINE
```

In the definition part of the tool, I added the two lines in **Listing 19Error! Reference source not found.** to indicate that the tool's options should be listed under "Unused Files" on the Options tab of the Configuration form, and that there are two options for this tool.

Listing 19. These two lines indicate that the tool has two options.

```
    .OptionTool = 'Unused Files'
    .OptionClasses = 'clsIncludeHome, clsChooseFolders'
```

The next step is to define clsUnusedFilesOptions, using Dynamic Forms. That code is shown in **Listing 20**; the Options page for Unused Files is shown in **Figure 8**.

Listing 20. This code specifies that the Options page for the Unused Files tool should contain a checkbox, a label and a combobox.

```
DEFINE CLASS clsUnusedFilesOptions AS Container

Procedure Init
  loRenderEngine = Execscript(_Screen.cThorDispatcher, 'Class= OptionRenderEngine')

  Text To loRenderEngine.cBodyMarkup Noshow Textmerge

    .Class = 'CheckBox'
    .Width = 300
    .Left = 25
    .WordWrap = .T.
    .Caption = 'Include project's home folder in folders to search'
    .cTool = 'Unused Files'
    .cKey = 'IncludeHome'
  |
    .Class = 'Label'
    .Width = 300
    .Left = 25
    .margin-top = 10
    .Caption = 'Where should the list of folders to search come from?'
  |
    .Class = 'Combobox'
    .Width = 300
    .Left = 30
    .RowSourceType = 1
    .RowSource = 'Current path,Project folders'
    .cTool = 'Unused Files'
    .cKey = 'ChooseFolders'
    .margin-top = -20
  ENDTEXT

  loRenderEngine.Render(This, 'Unused Files')

RETURN
ENDPROC

ENDDEFINE
```

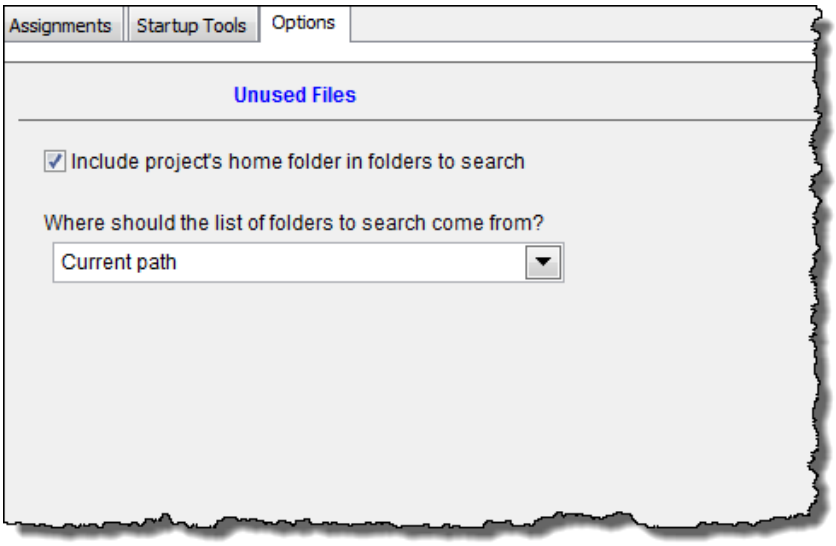

Figure 8. The code in Listing 20 generates this page to specify options for the Unused Files tool.

Ordinarily, Dynamic Forms spaces controls evenly as you go down the "page." When I first tested my options page, I found that the label was too far above the combobox; the three controls were evenly spaced. I looked at the Dynamic Forms documentation (http://vfpx.codeplex.com/wikipage?title=Dynamic%20Forms) and found the margin-top property that let me indicate that the margin above the combobox should be 20 pixels less than usual and that the label's should be 10 pixels more than usual. I also set the Left property for the combo to 30, rather than the 25 I used for the label, so the combo would be indented from the label.

The last step in implementing these options is to use their values in the tool code. Since both options affect the list of folders to check, but not the actual search process, all the changes are in the top portion of the tool's code. **Listing 21** shows the new code; it replaces the shorter block that populates the DirsToCheck cursor in Listing 4.

Listing 21. This code uses the two new options to determine which folders to search for unused files.

```
* Determine what list of directories to use
* based on option.
LOCAL cWhichFolders
cWhichFolders = UPPER(EXECSCRIPT(_Screen.cThorDispatcher, "Get Option=", ;
                      'ChooseFolders', 'Unused Files'))

IF m.cWhichFolders = 'PROJECT FOLDERS'
  CREATE CURSOR csrFolders (cFolder C(254))
ENDIF

oProject = _VFP.ActiveProject
* First, make a list of all files in project
LOCAL aProjFiles[oProject.Files.Count]

nCounter = 0
FOR EACH oFile IN oProject.Files
```

```
  nCounter = m.nCounter + 1
  aProjFiles[m.nCounter] = UPPER(oFile.Name)
  IF m.cWhichFolders = 'PROJECT FOLDERS'
    INSERT INTO csrFolders VALUES (JUSTPATH(oFile.Name))
  ENDIF
ENDFOR

CREATE CURSOR Unused (mFileName M)

* Now traverse directories
CREATE CURSOR DirsToCheck (mDirName M)

* If specified, include the project's home directory
LOCAL lIncludeProjectFolder
lIncludeProjectFolder = EXECSCRIPT(_Screen.cThorDispatcher, "Get Option=", ;
                                   'IncludeHome', 'Unused Files')

DO CASE
CASE m.cWhichFolders = 'CURRENT PATH'
  * First, make a list of directories
  * using current path
  nDirs = ALINES(aDirs, SET("Path"), 1, ";", ",")

  IF m.lIncludeProjectFolder
    INSERT INTO DirsToCheck VALUES (oProject.HomeDir)
  ENDIF

  FOR nDir = 1 TO m.nDirs
    INSERT INTO DirsToCheck VALUES (FULLPATH(aDirs[m.nDir], oProject.HomeDir + "\"))
  ENDFOR

CASE m.cWhichFolders = 'PROJECT FOLDERS'
  INSERT INTO DirsToCheck ;
    SELECT distinct cFolder FROM csrFolders

  IF m.lIncludeProjectFolder
    SELECT DirsToCheck
    LOCATE FOR UPPER(mDirName) == ALLTRIM(oProject.HomeDir)
    IF NOT FOUND()
      INSERT INTO DirsToCheck VALUES (oProject.HomeDir)
    ENDIF
  ENDIF

ENDCASE
```

With these changes, you can now decide whether to search based on the current path or the list of folders already included in the project, and whether to add the project's home folder to the search list, if it's not already there. This version of the tool is included in the session materials as Thor_Tool_Unused_Files_Take_2.PRG.

Working with Thor code

Thor is written entirely in VFP. A lot of code that's part of Thor is available to you for use in your tools. In addition, several aspects of Thor are designed to let you substitute your own code for the code that comes with Thor.

In this section of this paper, we'll look at Thor Plug-Ins, the Thor framework, and Thor Procs.

Thor Plug-Ins

There are a number of places where various Thor tools need particular services, and individual users may differ on how they want those services performed. This is similar to the need for options for Thor tools, but in this case, a particular service may apply to multiple tools or the number of different variations individual users want may make it difficult to handling the choices with options. Jim Nelson, the creator of Thor, describes it this way:

> "Of all the issues about PEM Editor and Thor functionality over the years, the one topic that aroused the most varied responses was how the list of variables in LOCAL statements (from tool Create Locals) should appear.
>
> "I was taken aback by this, as my own interest was only that all local variables be properly declared. It had not occurred to me that the actual structure of the LOCAL statements would be so critical to so many.
>
> "So, at first, I began adding a number of options to Create Locals to provide for as many variations as were suggested (see the options page for Create Locals), but soon gave up as I realized it was not possible to satisfy everybody's interest.
>
> "Thus, one of the earliest plug-in prgs was created. This plug-in, like all plug-ins, allows you to replace built-in behavior with your own."

You can see a list of Thor plug-ins by choosing Thor | More | Manage Plug-ins from the menu. Doing so opens the form shown in **Figure 9**. To modify a plug-in, click the Create button next to it in the form. Doing so opens a copy of the plug-in; if you save it, it's stored by default in the My Tools folder. That is, as with the tools that come with Thor, you can make changes to plug-ins without overwriting the original. Also as with tools, if there's a copy of a plug-in in the My Tools folder, Thor uses it; otherwise, it uses the original (which is generally built into PEMEditor.APP).

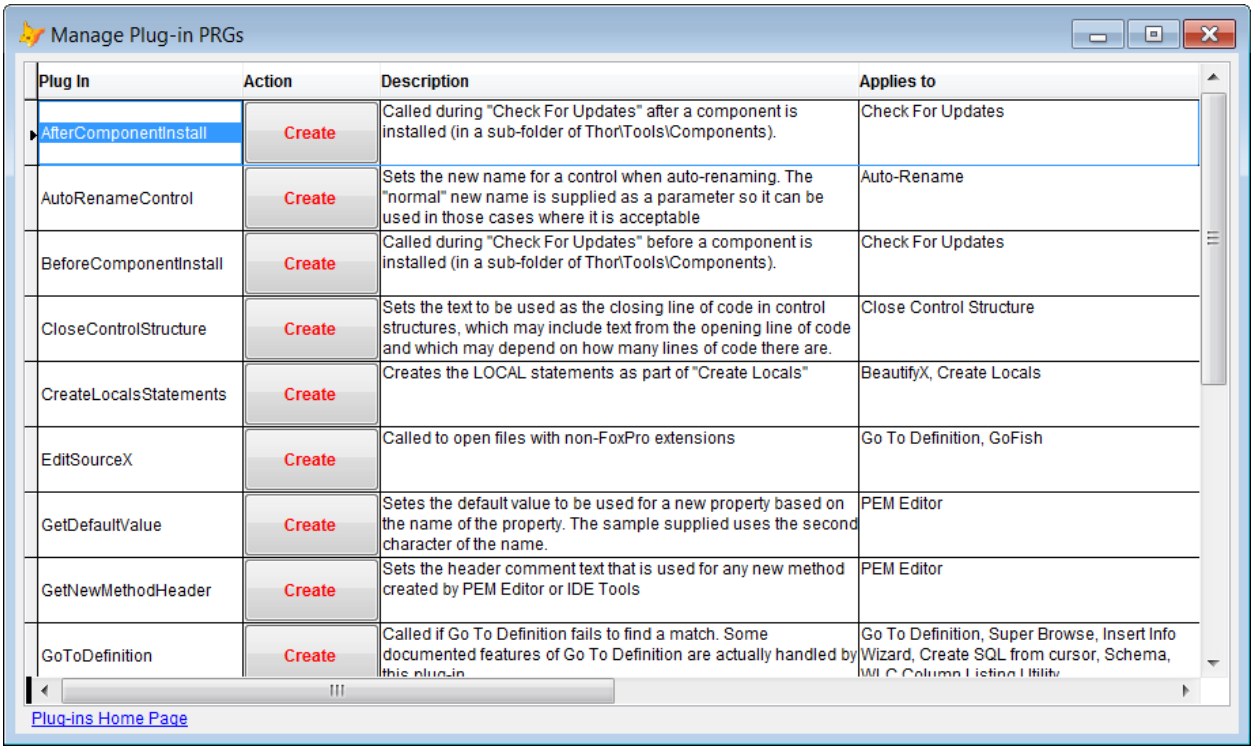

Figure 9. This form lets you review and manage Thor plug-ins.

In addition to the description for each plug-in in the Manage Plug-in PRGs form, the link at the bottom of the form opens the VFPX page for Thor Plug-ins. (As of this writing, that page is somewhat out of date, but still useful.)

You can create a link between a particular tool and one or more plug-ins, so that the tool's listing in the Thor Configuration form and the Launcher lets you see which plug-ins the tools uses.

To specify which plug-ins a tool uses, add the Plugins property to the tool definition, as in **Listing 22**. When that property has a non-empty value, a Plugins link appears on the tool's information page, as in **Figure 10**. When you click that link, it opens the Manage Plug-in PRGs form, showing only the listed plug-ins. **Figure 11** shows the plug-ins for BeautifyX.

Listing 22. BeautifyX uses the IsOperator and CreateLocalsStatements plug-ins.

```
        .PlugIns            = 'IsOperator, CreateLocalsStatements'
```

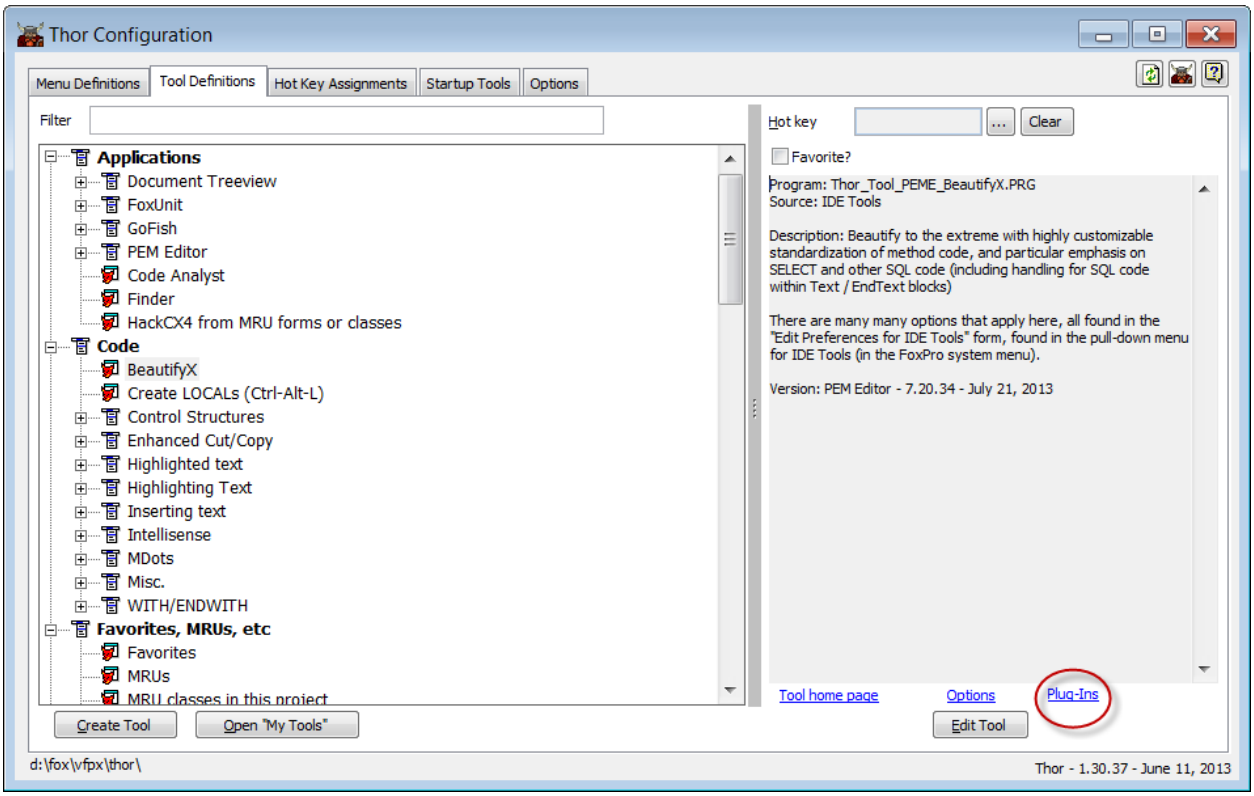

Figure 10. When the PlugIns property is populated in the Tool definition, a Plug-Ins link appears for the tool. Click it to see what plug-ins that tool uses.

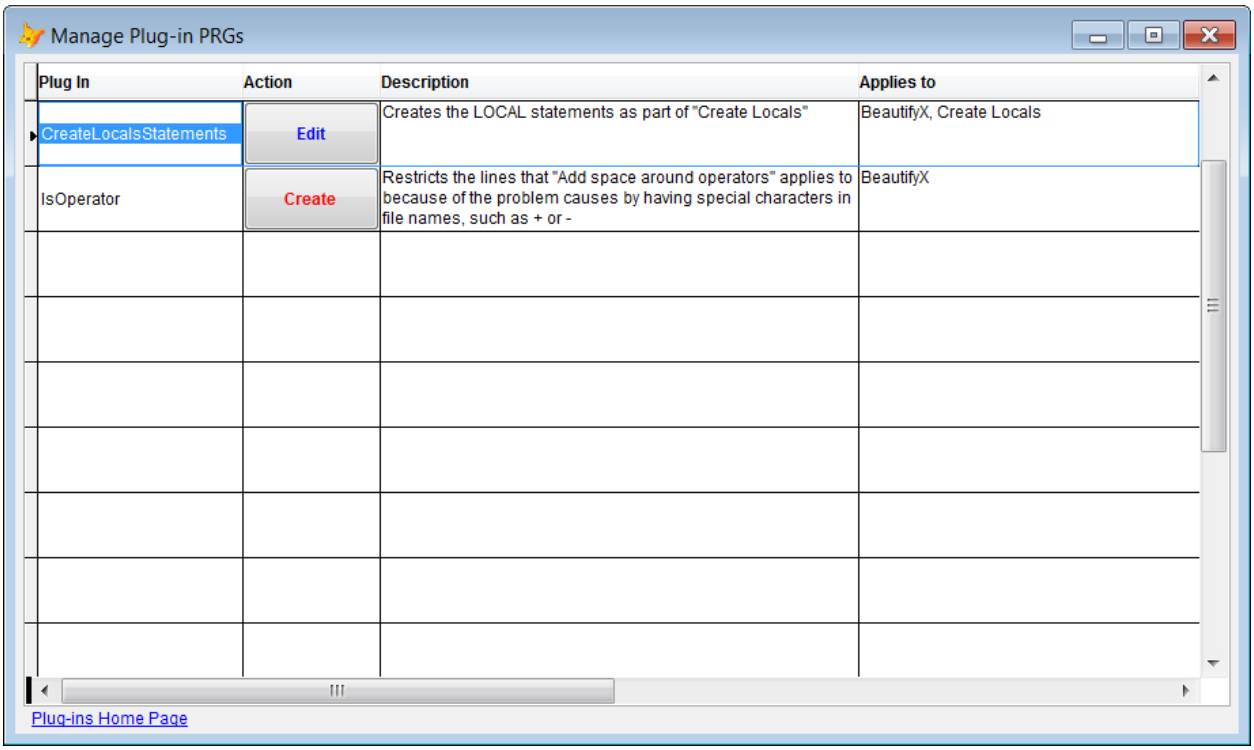

Figure 11. You can open this form showing only the plug-ins for a particular tool.

Creating a plug-in

Not only was Create Locals was the inspiration for the plug-in mechanism, but it's a good choice for demonstrating how plug-ins work. One of the options for this tool is to add an AS phrase to each declaration, specifying the data type. (While VFP uses the AS phrase only to aid IntelliSense, it's also helpful to people reading the code.)

However, that option only takes effect if the name of the variable begins with "l" (lower-case "L"). I don't use that notation in my code, since all variables are meant to be local unless I specify otherwise. I prefer to simply begin each variable name with a single character that indicates the data type. (For example, dBirth or lSuccess.)

So, I set out to modify the CreateLocalsStatements plug-in to be able to add an AS statement based on the assumption that the first character of the variable name specifies the type. Though it took me a while to get it right, it turned out that I only had to modify a few lines of the existing code. First, the code contains a loop through the list of variables found, using a CASE statement to determine how to show the variable. The first block of the CASE looks for the "Add AS Phrase" option and a variable starting with "l." The original code is shown in **Listing 23**. For my version, I simply commented out the AND portion of the case expression, as in **Listing 24**.

Listing 23. This code in the original CreateLocalsStatements plug-in restricts the "Add AS Phrase" option to variables beginning with "l."

```
Case llUseAsPhrase And llStartsWithL
  If lnType = 1
    lcVar      = AddASPhrase (lcName, lcASPhrase)
    lcLocalPhrase = lcLocalName
  Else
    Loop
  Endif
```

Listing 24. This version of the same code eliminates the requirement of starting with "l."

```
Case llUseAsPhrase && And llStartsWithL
  If lnType = 1
    lcVar      = AddASPhrase (lcName, lcASPhrase)
    lcLocalPhrase = lcLocalName
  Else
    Loop
  Endif
```

The remaining changes are in the AddASPhrase function. The original version is shown in **Listing 25**. Two changes are needed to base the type on the first character. First, the line that extracts the type letter needs to look at the first character rather than the second. In addition, the second case of the CASE statement, which skips over any name that doesn't start with "l" must be omitted. **Listing 26** shows the modified code.

Listing 25. This is the AddASPhrase function in the CreateLocalsStatements plug-in that comes with Thor.

```
Function AddASPhrase
```

```
  Lparameters lcName, lcASPhrase
  Local lcSuffix, lcVartype

  lcVartype = Upper (Substr (lcName, 2, 1))

  Do Case
    Case Not Empty (lcASPhrase)
      Return lcName + ' as ' + lcASPhrase
    Case lcName # 'l'
      Return lcName
    Case lcVartype = 'N'
      lcSuffix = ' as Number'
    Case lcVartype = 'I'
      lcSuffix = ' as Integer'
    Case lcVartype = 'O'
      lcSuffix = ' as Object'
    Case lcVartype = 'L'
      lcSuffix = ' as Boolean'
    Case lcVartype = 'C'
      lcSuffix = ' as String'
    Case lcVartype = 'U'
      lcSuffix = ' as Variant'
    Case lcVartype = 'D'
      lcSuffix = ' as Date'
    Case lcVartype = 'T'
      lcSuffix = ' as DateTime'
    Case lcVartype = 'Y'
      lcSuffix = ' as Currency'
    Otherwise
      lcSuffix = ''
  Endcase

  Return lcName + lcSuffix

Endfunc
```

Listing 26. This version of the AddASPhrase function uses the first character of the variable to determine the type.

```
Function AddASPhrase
  Lparameters lcName, lcASPhrase
  Local lcSuffix, lcVartype

  lcVartype = Upper (Left(lcName, 1))

  Do Case
    Case Not Empty (lcASPhrase)
      Return lcName + ' as ' + lcASPhrase
*!*      Case lcName # 'l'
*!*        Return lcName
    Case lcVartype = 'N'
      lcSuffix = ' as Number'
    Case lcVartype = 'I'
      lcSuffix = ' as Integer'
    Case lcVartype = 'O'
```

```
      lcSuffix = ' as Object'
    Case lcVartype = 'L'
      lcSuffix = ' as Boolean'
    Case lcVartype = 'C'
      lcSuffix = ' as String'
    Case lcVartype = 'U'
      lcSuffix = ' as Variant'
    Case lcVartype = 'D'
      lcSuffix = ' as Date'
    Case lcVartype = 'T'
      lcSuffix = ' as DateTime'
    Case lcVartype = 'Y'
      lcSuffix = ' as Currency'
    Otherwise
      lcSuffix = ''
  Endcase

  Return lcName + lcSuffix

Endfunc
```

With these changes, the "All variables use 'AS datatype' phrase" option is useful to me.

The Thor Framework

While you can write a new tool with standard VFP code (as in the Unused Files tool), Thor offers a large library of capabilities that make it easier to write tools. The Thor Framework gives you access to several classes, as well as some standard items you may want.

To access the Thor Framework, choose Thor | More | Thor Framework from the menu. A window opens, containing code you can cut and paste into your tool code, as well as links to home pages for any framework items or tools that have them. **Figure 12** shows part of the Thor Framework. This listing includes code to run various other Thor tools that link into the Thor menu; as you can see, I have GoFish and the Object Inspector installed in my Thor menu. The Thor Framework shows how you can, for example, make the GoFish search engine available to your tools.

The Thor Framework is smart enough to show the correct path for your installation, so that IntelliSense works. (So Figure 12 shows where the files are located on my computer.) Since VFP only cares about the AS portion of a LOCAL declaration for IntelliSense, having the wrong path isn't fatal. But IntelliSense is useful when working with the Framework, so when you're modifying somebody else's Thor tool code, you may want to replace any LOCAL statements with the ones from the Framework . Unfortunately, many of the tools in the Thor Repository were written before this was understood, so they don't use the AS phrase.

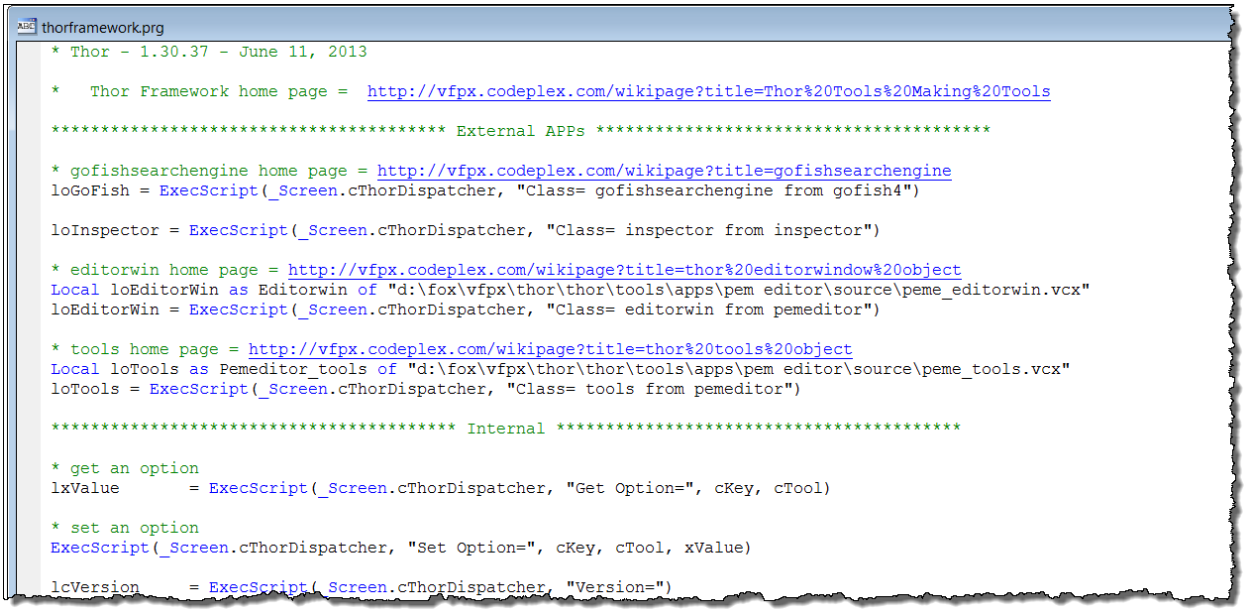

Figure 12. The Thor Framework lets you take advantage of code in Thor in your tools.

You can find an overview of the Thor Framework at http://tinyurl.com/cney2dz. There's detailed information in the chapter "Creating Thor Tools" of "*VFPX: Open Source Treasure for the VFP Developer.*"

The Thor Framework includes a set of classes, either built into Thor or built into other VFPX tools (like PEM Editor). In addition, any Thor tool can "publish" a class for inclusion in the Thor framework. For instance, the tool for GoFish includes the line in **Listing 27**. (This code is on a single line in the tool, but wraps here. In addition, I removed a trailing comment that made it harder to understand the line.)

Listing 27. Thor tools can make code available to the Thor framework.

```
.Classes = 'loGoFish = gofishsearchengine of
lib\gofishsearchengine.vcx|http://vfpx.codeplex.com/wikipage?title=GoFishSearchEngine
'
```

Everything up to the vertical bar is the class definition. Everything after the vertical bar is the portion that shows as a comment in the Framework. You can actually include multiple classes in the Classes property; just separate them with commas. **Listing 28** shows the Classes line for PEM Editor, which makes two of its classes available to the Thor Framework. (Again, this code is on a single line in the tool, but wraps here.)

Listing 28. PEM Editor makes two classes available to the Thor Framework, Tools and EditorWin.

```
.Classes = 'loTools = Tools of
PEME_Tools|http://vfpx.codeplex.com/wikipage?title=Thor%20Tools%20Object, loEditorWin
= EditorWin of PEME_EditorWin|http://vfpx.cod
   eplex.com/wikipage?title=Thor%20EditorWindow%20Object'
```

The window that opens when you choose the Thor Framework gives you the code you need to access those classes. There are four core classes, shown in **Table 1**. There's a documentation page for each class on the VFPX site. The Thor Framework listing provides a link to the documentation.

Table 1. The Thor Framework provides access to a set of classes that simplify tool writing.

Class	Purpose
EditorWindow	Provides methods to access and modify IDE windows and to access and modify text in the current code editing window.
Tools	Provides methods (used by PEM Editor) to manipulate forms and classes.
ContextMenu	Provides methods used to create pop-up menus and sub-menus.
FormSettings	Provides methods to save and restore form properties, such as size and position.

There's one additional piece of code that isn't technically part of the Thor framework, but is used by a large number of Thor tools. PEMEditor_StartIDETools provides access to the "IDE tools" that were bundled into PEM Editor. Calling it creates an instance of the PEM Editor engine, stored in a variable named _oPEMEditor. Once you create it, you can access the various tools through objects it contains. For example, _oPEMEditor has an oUtils property; the object it references has a method, EditParentClasses, which runs the Edit Parent and Containing Classes tool.

As Figure 12 indicates, the way to use these classes is to instantiate them using an ExecScript() call to the Thor dispatcher, _Screen.cThorDispatcher. Save a reference to the instantiated object and call its methods.

The same technique, calling ExecScript() passing _Screen.cThorDispatcher as the first parameter, gives you access to some other features of the Thor Framework. Perhaps the most useful is the ability to find the path to any file that's in the Thor hierarchy; **Listing 29** demonstrates.

Listing 29. You can use the Thor Framework to find the path to any file in the Thor hierarchy.

```
lcFileName    = ExecScript(_Screen.cThorDispatcher, "Full Path=" + m.cFileName)
```

Table 2 shows the non-class capabilities of the Thor framework. The second parameter to ExecScript() determines each ability. Be aware that spaces are significant in that parameter.

Table 2. The Thor framework provides access to a variety of information.

Second parameter to ExecScript()	Result
"Get Option="	Returns the value of the specified option. You must also pass the key for the option and the name of the tool. (See "Accessing an option," earlier in this paper.)
"Set Option="	Sets the value of the specified option. You must also pass the key for the option, the name of the tool and the new value. (See "Using invisible options," earlier in this paper.)
"Version="	Returns the Thor version number.

Second parameter to ExecScript()	Result
“Tool Folder=”	Returns the full path to the Thor Tools folder.
“Full Path=” + ToolName	Returns the full path and filename for the specified tool.
“DoDefault()”	Runs the default version of the Thor Proc (see “Thor Procs,” later in this paper) or Thor Tool in which the call occurs. Lets you extend Thor Procs and Tools without duplicating code.
“Thor Template Code=”	Returns the default code for a new Thor Tool, that is, the code you see when you start creating a new Tool.
“Run”	Runs Thor.
“Edit”	Opens the Thor Configuration form.
“Clear Hotkeys”	Clears all Thor hotkeys.

There are also several calls you can make that are based on classes that are part of Thor. **Table 3** lists them.

Table 3. A few items in the Thor Framework call methods of Thor classes.

Second parameter to ExecScript()	Thor class	Result
“ToolInfo=”	Thor_Utils.VCX	Returns the ThorInfo object for the specified tool, which you must pass as the third parameter. The ThorInfo object is the object created with the properties specified in the top part of a Thor Tool.
“Thor Register=”	Thor_Utils.VCX	Returns an object used to register applications with Thor. Used by GoFish and others.
“Thor Engine=”	Thor.VCX	Returns a reference to the main Thor engine.

The tools that come with Thor make extensive use of the Thor Framework. **Listing 30** shows the ToolCode procedure for the Add MDots tool. It uses the EditorWindow class to extract the code to be processed and the “Get Option=” technique to look up the user’s choices. This code also calls the Thor Proc Thor_Proc_AddMDotsMultipleProcs; see “Thor Procs,” later in this paper for an explanation of Thor Procs.

Listing 30. The Add MDots tool uses several features of the Thor framework.

```
Local lcClipText, lcNewCliptext, lcOldClipText, lnMDotsUsage, ;
      lnSelEnd, lnSelStart, loEditorWin

* get the object which manages the editor window
* see  http://vfpx.codeplex.com/wikipage?title=Thor%20EditorWindow%20Object
loEditorWin = Execscript (_Screen.cThorDispatcher, 'class= editorwin from pemeditor')
* locate the active editor window; exit if none active
If loEditorWin.GetEnvironment(25) <= 0
  Return
Endif

lcOldClipText = _Cliptext
lnSelStart    = loEditorWin.GetSelStart()
lnSelEnd    = loEditorWin.GetSelEnd()
If lnSelStart = lnSelEnd
  loEditorWin.Select(0, 1000000)
Endif
```

```
* copy highlighted text into clipboard
loEditorWin.Copy()
lcClipText = _Cliptext

lnMDotsUsage  = ExecScript(_Screen.cThorDispatcher, "Get Option=", ;
                           'MDots Usage', 'MDots')
lcNewCliptext = Execscript (_Screen.cThorDispatcher, ;
                            'Thor_Proc_AddMDotsMultipleProcs', lcClipText, ;
                            lnMDotsUsage = 3)

************************************************************
* This final block of code pastes in the modification (in <lcNewCliptext>)
loEditorWin.Paste (lcNewCliptext)
_Cliptext = lcOldClipText
loEditorWin.Select (lnSelStart, lnSelStart)
loEditorWin.SetInsertionPoint (lnSelStart)

Return
```

The Unused Files tool described earlier in this paper works only when you have a project open. One of the cool features of many Thor tools is that they can see what's at the cursor position and operate on that item. That would be a handy capability for this tool—if there's no open project, then find the name at the cursor position and attempt to open that project.

To figure out how that capability was provided, I poked around in the code for Thor tools that have it. As described in "Editing your Thor tools," earlier in this paper, to see how any Thor tool is implemented, open the Thor Configuration form and switch to the Tool Definitions page. Select the tool you're interested in the treeview and click the Edit Tool button. For tools you haven't modified, the form shown in **Figure 3** appears. If all you want to do is see how the tool works, choose the second button, "View this file in Read-Only mode."

In the code for the SuperBrowse tool, I found the lines in **Listing 31**. As you can see, it uses the Tools class from the Thor Framework.

Listing 31. The code that implements the SuperBrowse tool uses this code to determine what table to browse.

```
* tools home page = http://vfpx.codeplex.com/wikipage?title=thor%20tools%20object
loTools = Execscript ( _Screen.cThorDispatcher, 'class= tools from pemeditor')
loTools.UseHighlightedTable ( Set ('Datasession'))
```

I dug into the PEM Editor's source to find the UseHighlightedTable method, which I found in the PemEditor_Tools class of PEME_Tools.Vcx (which is the implementation of the framework's Tools class). Eventually, I found the line of code in **Listing 32**. Since the method name implies that it only grabs the highlighted text, I tested to confirm that the method, in fact, picks up the entire word where the cursor is positioned.

Listing 32. This line of code, used by the SuperBrowse tool, reads the text under the cursor.

```
lcAlias = This.oUtils.oIDEx.GetCurrentHighlightedText()
```

The next step was to change the code for my Unused Files tool.

I copied the three lines from the framework that instantiate the Tools class and pasted them into the appropriate place in the ToolCode procedure. Once the Tools class is instantiated, I can use it to get the word under the cursor, and then try to open a project with that name. The relevant portion of the modified ToolCode procedure is shown in **Listing 33**.

Listing 33. Replace the code to check whether a project is open with this code to allow the Unused Files tool to work on the project whose name is under the cursor.

```
IF TYPE("_VFP.ActiveProject") = "U"
  * tools home page = http://vfpx.codeplex.com/wikipage?title=thor%20tools%20object
  LOCAL loTools AS Pemeditor_tools OF ;
    "d:\fox\vfpx\thor\thor\tools\apps\pem editor\source\peme_tools.vcx"
  loTools = EXECSCRIPT( _SCREEN.cThorDispatcher, "Class= tools from pemeditor")

  lcText = loTools.oUtils.oIDEx.GetCurrentHighlightedText()

  lProjWasOpen = .F.

  IF NOT EMPTY(m.lcText)
    TRY
      MODIFY PROJECT (lcText) NOWAIT
      lSuccess = (TYPE("_VFP.ActiveProject") <> "U")
    CATCH
      lSuccess = .F.
    ENDTRY
  ELSE
    lSuccess = .F.
  ENDIF
ELSE
  lSuccess = .T.
  lProjWasOpen = .T.
ENDIF

IF NOT m.lSuccess
  MESSAGEBOX("No active project", 0+48, "Unused files")
  RETURN
ENDIF
```

This version of the tool is included in the session materials as Thor_Tool_Unused_Files_Take_3.PRG.

In testing, I found that this code works only when the specified project is in the path and when the cursor is over the project name. If the cursor is positioned elsewhere in the path to the project, the code fails. Fortunately, it turns out that there's a better way to do this using Thor Procs.

Thor Procs

In addition to the Thor Framework, Thor also includes a set of programs called Thor Procs. These live in the Tools\Procs folder of your Thor installation and have names that begin "Thor_Proc_." The rest of the name describes the functionality.

To use a Thor Proc, call it using an ExecScript() call, as in **Listing 34**, passing as many parameters as needed.

Listing 34. Thor Procs are called using the ExecScript() notation typical of Thor.

```
ExecScript(_Screen.cThorDispatcher, 'Thor_Proc_SomeThorProc', ;
           'MyParameter1', 'MyParameter2', …)
```

Thor Procs can return a result; the mechanism for doing that also uses ExecScript, as in **Listing 35**.

Listing 35. Thor Procs return a result using the ExecScript() notation.

```
Return Execscript (_Screen.cThorDispatcher, 'Result=', lcResult)
```

The Thor Procs, as of this writing, are listed in **Table 4**. Among other tasks, the whole Thor Update process is implemented using Thor Procs.

Table 4. Thor Procs provide callable functionality for Thor Tools.

Proc name	Purpose
AddMDotsMultipleProcs	Adds mdots to the code in one or more routines.
AddMDotsSingleProc	Adds mdots to the code in a single routine.
AfterComponentInstall	Placeholder for code to run after installing or updating a VFPX component.
AfterInstall	Placeholder for code to run after installing or updating a VFPX tools.
BeautifyOption	Looks up options for Beautify stored in the VFP Resource file.
BeforeComponentInstall	Placeholder for code to run before installing or updating a VFPX component.
BeforeInstall	Placeholder for code to run before installing or updating a VFPX tool.
Check_For_Updates	Looks for updates to Thor and any installed tools.
CheckCodeBlockForBadReturns	Looks for RETURN statements within WITH/ENDWITH blocks.
CheckFileForBadReturns	Looks for RETURN statements within WITH/ENDWITH blocks. Intended to work on both PRGs and VCXs, but currently applies only to PRGs.
CheckForUpdate	Wraps the GetAvailableVersionInfo Thor Proc. Appears to be unused.
CheckInternetConnection	Checks whether there's an active Internet connection.
CloseTemps	Tracks open tables at the start of a process and closes those opened by the process.
CommentText	Class for accessing and modifying the highlighted text in a code window.
ConvertTimeStamp	Converts the timestamp used in VFP's "X files" into one of three more readable formats.
DBCTables	Creates a dropdown list of tables in a database, along with those in the data environment of the form being edited.

Proc name	Purpose
DownloadAndExtractToPath	Downloads a file from a specified URL and unzips it into a specified folder.
DownloadAndInstallUpdates	Downloads and installs updates either to Thor or to a specified list of VFPX tools.
DownloadFileFromURL	Downloads a file from a specified URL and stores it with a specified name.
DynamicForm	Thor-specific version of Dynamic Form tool. Used to implement the "OptionRenderEngine" class.
EditProc	Creates a new Thor tool program or opens an existing Thor tool program for editing.
Expand_Bitly_Url	Converts a bit.ly shortened URL into the full URL.
ExtractFiles	Copies files from a specified ZIP file or folder to a specified folder.
ExtractFilesFromZip	Extracts files from a specified ZIP file to a specified folder.
ExtractToPath	Extracts the files in a specified ZIP file to first the Temp folder, then a specified destination folder.
FindNonCodeBlocks	Creates a cursor of blocks within a code block that are not code (such as comments).
GenerateSCCFilesOnProject	Applies the VFPX tool SCCTextX to all the code files in a project.
GetAvailableVersionInfo	Retrieves information about the currently available version for a VFPX project.
GetDataEnvironmentList	Fills an array with a list of the tables in the Data Environment of the Form Designer.
GetDBCTablesList	Fills an array with a list of the tables in a specified database.
GetFieldNames	Converts the array created by AFIELDS() into an array in the format used by IntelliSenseX.
GetFieldsFromObjectName	Retrieves the list of fields for an object representation of a table.
GetFilesFromProjectForSCC	Returns a cursor listing the files in a project that need conversion to text for source control purposes.
GetFullStartPositions	Creates a collection showing the start position for each item (procedure, function, class, etc.) in a code block.
GetHackCX	Returns the full path to HackCX, if it's installed.
GetHighlightedText	Returns the text at the cursor position, accepting a parameter to indicate what kind of thing to look for. See the example later in this section for details.
GetISXOptions	Returns an object with a property for each IntelliSenseX option.
GetMaxTimeStamp	Returns, as a datetime, the latest timestamp from a specified file.
GetNamesFromCodeBlock	Retrieves a list of all the variable and parameter names used in a code block, with an indication of the scope of the name.
GetNormalIndentation	Returns one instance of the user's specified indentation string (either a tab or some number of spaces).
GetParameters	Returns the list of parameters in a code block.
GetPrgBasedClassPEMs	Retrieves a list of PEMs for classes defined in a code block.
GetProcedureStartPositions	Retrieves a list of the start positions and length of each procedure, function or method in a code block.
GetProcFromPRG	Returns the procedure that includes a specified position in a code block.
GetPropertyList	Retrieves a list of the PEMs in a specified object. An optional parameter indicates which kinds of members to include.
GetRegistryOption	Retrieves the value of an option from the VFP section of the Registry.
GetUpdateList	Retrieves the list of programs for updating VFPX tools through the Thor Update Process.

Proc name	Purpose
GetUpdaterObject	Creates and returns the object used to drive the Thor Update Process.
HighlightControlStructure	Highlights the appropriate control structure (by calling the appropriate PEM Editor method).
IntellisenseXTimer	Manages a timer used by IntelliSenseX.
ISX	Is the main program for IntelliSenseX (based on Christof Wollenhaupt's ISX).
ListTablesInPath	Builds a list of tables in the current path for use by IntelliSenseX.
MessageBox	Displays a message box and returns the result.
NewSessionObject	Instantiates a specified class, giving it its own data session.
OpenExplorer	Opens Windows Explorer, optionally pointed at a particular folder or file.
OpenFolder	Opens Windows Explorer, pointed at a particular folder.
OpenOptionsPage	Opens the Thor Configuration form to the Options page, pointed at a specified tool.
PEME_EnhancedCutCopy	Implements the Enhanced Cut/Copy tools.
PEME_ExtractToMethod	Implements the Extract To Method tool.
PEME_GoToDefinition	Implements advanced features of the Go To Definition tool.
PEME_GoToDefPRGxClasses	Searches for a specified method definition in a PRG-based class, as part of the Go To Definition tool.
Registry	Contains the Registry class that comes with VFP.
RelativePath	Returns the relative path between a specified file and a specified folder.
Repository_FoxrefPatch	Substitutes the Go To Definition tool and GoFish for the View Definition and Look Up Reference items in the Edit window context menu.
SCCTextX	Converts VFP's binary formats (SCX, VCX) to text for use with source control systems. An enhanced version of SCCText.PRG that comes with VFP.
SelectText	Highlights the specified text.
SetLibrary	Adds a specified FLL to the list of libraries available. Downloads the file if necessary.
SetRegistryOption	Sets the value of an option in the VFP section of the Registry.
Shell	Runs a specified file using the Windows shell.
ShowErrorMessage	Displays an error message.
SuperBrowse	Displays the SuperBrowse form. If the form is already open, activates it.
UpdateTimestampsOnFolder	Modifies the timestamp for each SCX, VCX and FRX file in a folder to the latest timestamp of any record in the file, unless there's a corresponding source control file with a later date than any of them. Optionally calls itself recursively to process sub-folders.
UpdateTimeStampsOnProjectFiles	Modifies the timestamp for each SCX, VCX and FRX file in a project to the latest timestamp of any record in the file, unless there's a corresponding source control file with a later date than any of them.
UpdateWaitWindow	Displays a WAIT window with a specified message.
WriteToCFULog	Adds a message to Thor's Check For Updates log.
ZDEF	Retrieves a list of defined constants. Based on VFP's ZDEF IntelliSense script.

Many Thor Procs are quite specific, providing functionality for a specific tool. Others are wrappers around generic functionality. But a few are more broadly useful.

The Thor Proc GetHighlightedText returns the text at the current cursor position, and can handle filenames including a path. This provides a better solution to allow the Unused Files tool to work by just clicking into the name of the project. The Proc accepts a single parameter to indicate what to return; **Table 5** shows the acceptable values for the parameter.

Table 5. Pass a string to Thor_Proc_GetHighlightedText to indicate what text to return.

Parameter value	Returns
Empty (omitted)	The currently highlighted text.
"Name"	The word the cursor is currently positioned in.
"File Name"	The filename the cursor is currently positioned in, including path.
"Object Name"	The object name the cursor is currently positioned in, including any objects specified by WITH statements.

Listing 36 shows the revised part of the code that handles opening the file, if necessary. If there's no active project, the GetHighlightedText Proc is called to retrieve the full name and path. If anything comes back and there is such a file, we attempt to open the project. If we're unsuccessful in any way, we bail out.

Listing 36. Using the Thor Proc GetHighlightedText, we can pick up the full name and path of the project in the Get Form Classes tool.

```
IF TYPE("_VFP.ActiveProject") = "U"
  lcText = ExecScript(_Screen.cThorDispatcher, "Thor_Proc_GetHighlightedText", ;
                      "File Name")

  lProjWasOpen = .F.

  TRY
    IF NOT EMPTY(m.lcText) AND FILE(m.lcText)
       MODIFY PROJECT (lcText) NOWAIT
       lSuccess = (TYPE("_VFP.ActiveProject") <> "U")
   ELSE
      lSuccess = .F.
   ENDIF
  CATCH
    lSuccess = .F.
  ENDTRY
ELSE
  lSuccess = .T.
  lProjWasOpen = .T.
ENDIF

IF NOT m.lSuccess
  MESSAGEBOX("No active project", 0+48, "Unused files")
  RETURN
ENDIF
```

The final version of the Unused Files tool is included in the downloads for this session as Thor_Tool_Unused_Files.PRG. You can copy it into the Tools folder of your Thor installation to add it.

Think about sharing

With all the code and customization Thor provides, you can build some pretty impressive tools that may be useful not only to you, but to others in the VFP community. If you've built such a tool, consider submitting it to the Thor Repository. The criteria for doing so are described in the VFPX page for the Repository: http://vfpx.codeplex.com/wikipage?title=Thor%20Repository.

To submit a tool, send an email with the tool code attached to FoxProThor@googlegroups.com.

Get involved or get help

You can get involved with Thor or just get help with any problems you run into in Thor's Google Group, accessible from the Thor menu (Thor | Forums | Thor). There, you'll find discussion about how Thor and its tools should work, bugs that people have found, new Thor Tools and more.

In addition, the book "VFPX: Open Source Treasure for the VFP Developer" has four chapters devoted to Thor, including a complete catalog of the tools that came with Thor at the time the book was written. (Several have been added since.) You can buy the book at http://www.foxrockx.com/GetVFPX.htm.