Try Thor's Terrific Tools

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

The VFPX project, Thor, includes dozens of tools to aid in development. In this session, we'll look at some of what Thor has to offer. The session will explore a number of Thor tools, including Document View, Create Locals, Compare Objects, and much more. We'll also see how to make any Thor tool available with a keyboard shortcut. We'll also look at how to add your own tools to Thor and, if time permits, how to set up user preferences for a tool.

One of the things that makes Visual FoxPro such a great tool for developing software is the open architecture that makes it easy to create developer tools. It's rare to find an experienced VFP developer who hasn't written at least one tool to automate some task in the IDE. Some people have a whole menu pad's worth of developer tools (and in fact, being able to add a menu pad is one example of VFP's open architecture).

The VFPX website was created to allow VFP developers to share tools, and it now houses quite a few developer tools (along with a bunch of components meant to be used in VFP apps). But it's not a great way to share little tools. What's a little tool? Something that takes just a few lines of code, perhaps with no user interaction needed. Something where creating a whole VFPX project would be overkill.

As the VFPX tool PEM Editor was reaching maturity, Jim Nelson and Matt Slay, its principal authors and designers, found that there were lots of little tools they wanted and they started adding them to PEM Editor. But many of these little tools weren't really relevant to managing properties, events and methods of forms and classes. PEM Editor just proved to be a handy way of distributing them.

Eventually, they realized that what was really needed was a tool for managing tools, and Thor was born. Thor is a tool designed to let you manage developer tools; it comes with a whole set of tools, but is extensible so you can add your own developer tools, as well as those you get from others. Thor allows you to assign hot keys to any installed tool, as well as to create custom pads on the VFP menu and custom pop-up menus accessed by hot keys.

In this session, we'll look at a number of the tools that come with Thor to show you why you want to bother changing the way you work. Since most of the tools operate on code in the IDE, we'll need to demonstrate on some programs, forms and classes. As much as possible, I'll use code that comes with VFP, such as classes from the FFC (FoxPro Foundation Classes).

Getting started with Thor

To use Thor, all you have to do is download it from VFPX and install it. The VFPX website includes instructions for installing Thor at http://vfpx.codeplex.com/wikipage?title=Thor%20Install&referringTitle=Thor%20Help. Once you've done so, you'll see two new pads on the VFP menu (see Figure 1) and the Thor Configuration form will be open. The Thor pad contains items for managing Thor itself. The Thor Tools pad contains the tools that come with Thor; you can add tools, as well as modify and remove the built-in tools. You can also specify that any tool should run when Thor starts, and assign a hot key to any tool. All of those things are done using the Thor Configuration form, available from the Thor menu pad.

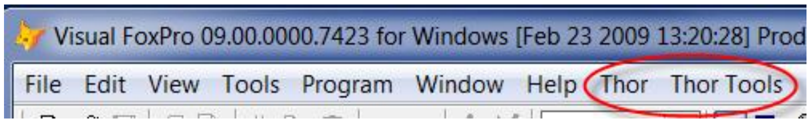

Visual FoxPro 09.00.0000.7423 for Windows [Feb 23 2009 13:20:28] Prod
File Edit View Tools Program Window Help Thor Thor Tools

Figure 1. Thor adds two pads to the VFP system menu.

To ensure that Thor opens each time you open VFP, add one line of code, shown in Listing 1, to your VFP start-up program. The number you pass as a parameter determines how often the Thor Update process runs; pass a number of days. Thor Update helps you keep Thor and other tools from VFPX up-to-date.

Listing 1. This line of code in your VFP start-up program checks for VFPX updates weekly.

```
DO D:\Fox\VFPX\Thor\Thor\RunThor.PRG WITH 7
```

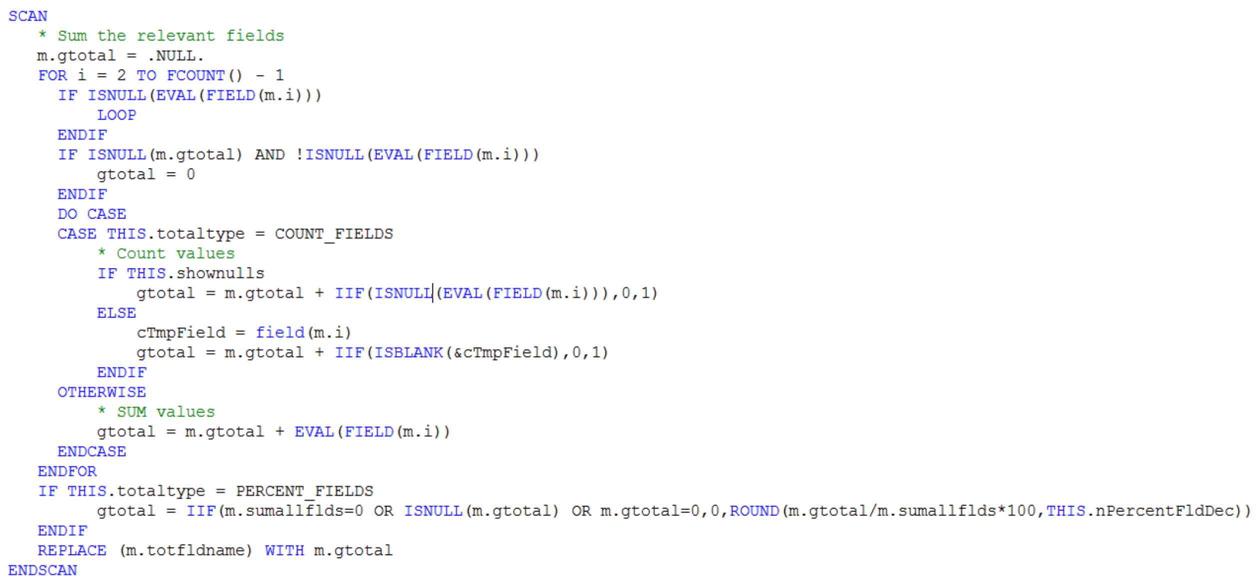
Once Thor is installed and set to run each time you start VFP, you're ready to start exploring the tools it provides. Much of this document explores a subset of those tools, the ones I find most compelling. You may find that others are your favorites. I'll also show you how to make any Thor tools you want easily accessible.

Highlight Control Structure

Menu: Code | Control Structures | Highlight Control Structure

I work a lot with other people's code, that is, code originally written by another developer. Sometimes, even beautifying the code isn't enough to help me grasp its structure. Thor's Highlight Control Structure tool is handy when I'm looking at a particular code block and trying to understand it. It highlights the entire structure where the cursor is found, whether it's IF-ENDIF, FOR-ENDFOR, DO CASE, SCAN-ENDSCAN, DO WHILE, TEXT-ENDTEXT or TRY-CATCH. If you run the tool a second time, it highlights the structure containing the one you already highlighted. Subsequent uses continue to work their way outward.

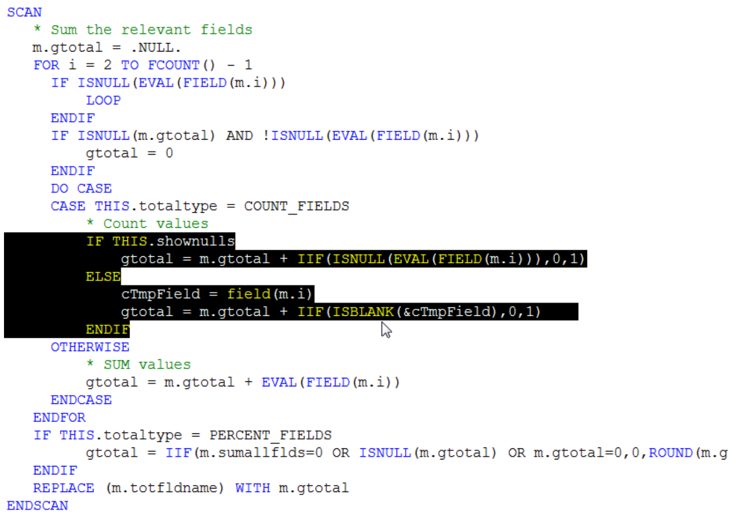
For example, Figure 2 shows a block of code from VFPXTAB.PRG. The cursor is positioned on an assignment statement that's inside an IF block (between "ISNULL" and its open parenthesis). The IF block is inside a CASE statement, which is contained in a FOR loop. The FOR loop is inside a SCAN loop.



```
SCAN
   * Sum the relevant fields
   m.gtotal = .NULL.
   FOR i = 2 TO FCOUNT() - 1
     IF ISNULL(EVAL(FIELD(m.i)))
         LOOP
     ENDIF
     IF ISNULL(m.gtotal) AND !ISNULL(EVAL(FIELD(m.i)))
         gtotal = 0
     ENDIF
     DO CASE
     CASE THIS.totaltype = COUNT_FIELDS
         * Count values
         IF THIS.shownulls
             gtotal = m.gtotal + IIF(ISNULL(EVAL(FIELD(m.i))),0,1)
         ELSE
             cTmpField = field(m.i)
             gtotal = m.gtotal + IIF(ISBLANK(&cTmpField),0,1)
         ENDIF
     OTHERWISE
         * SUM values
         gtotal = m.gtotal + EVAL(FIELD(m.i))
     ENDCASE
   ENDFOR
   IF THIS.totaltype = PERCENT_FIELDS
         gtotal = IIF(m.sumallflds=0 OR ISNULL(m.gtotal) OR m.gtotal=0,0,ROUND(m.gtotal/m.sumallflds*100,THIS.nPercentFldDec))
   ENDIF
   REPLACE (m.totfldname) WITH m.gtotal
ENDSCAN
```

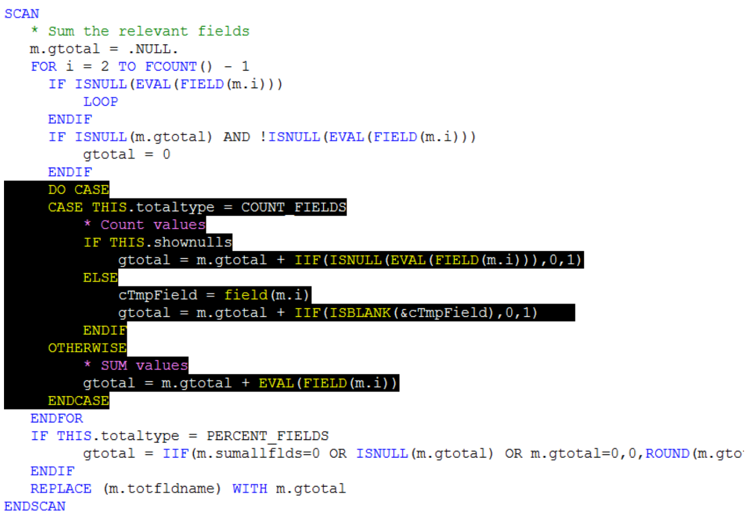
Figure 2. This block of code, drawn from VFPXTAB.PRG, has an IF inside a CASE, inside a FOR loop, inside a SCAN loop.

Figure 3 shows the result of using Highlight Control Structure, while Figure 4 shows the code after the second application of Highlight Control Structure. Using the tool a third time would highlight the entire FOR loop, and a fourth use highlights the whole SCAN. In fact, this entire block of code is inside another IF statement, and a fifth use of Highlight Control Structure highlights that IF.



```
SCAN
   * Sum the relevant fields
   m.gtotal = .NULL.
   FOR i = 2 TO FCOUNT() - 1
     IF ISNULL(EVAL(FIELD(m.i)))
         LOOP
     ENDIF
     IF ISNULL(m.gtotal) AND !ISNULL(EVAL(FIELD(m.i)))
         gtotal = 0
     ENDIF
     DO CASE
     CASE THIS.totaltype = COUNT_FIELDS
         * Count values
         IF THIS.shownulls
             gtotal = m.gtotal + IIF(ISNULL(EVAL(FIELD(m.i))),0,1)
         ELSE
             cTmpField = field(m.i)
             gtotal = m.gtotal + IIF(ISBLANK(&cTmpField),0,1)
         ENDIF
     OTHERWISE
         * SUM values
         gtotal = m.gtotal + EVAL(FIELD(m.i))
     ENDCASE
   ENDFOR
   IF THIS.totaltype = PERCENT_FIELDS
         gtotal = IIF(m.sumallflds=0 OR ISNULL(m.gtotal) OR m.gtotal=0,0,ROUND(m.g
   ENDIF
   REPLACE (m.totfldname) WITH m.gtotal
ENDSCAN
```

Figure 3. Using Highlight Control Structure on the code in Figure 2 highlights just the IF statement where the cursor was positioned.



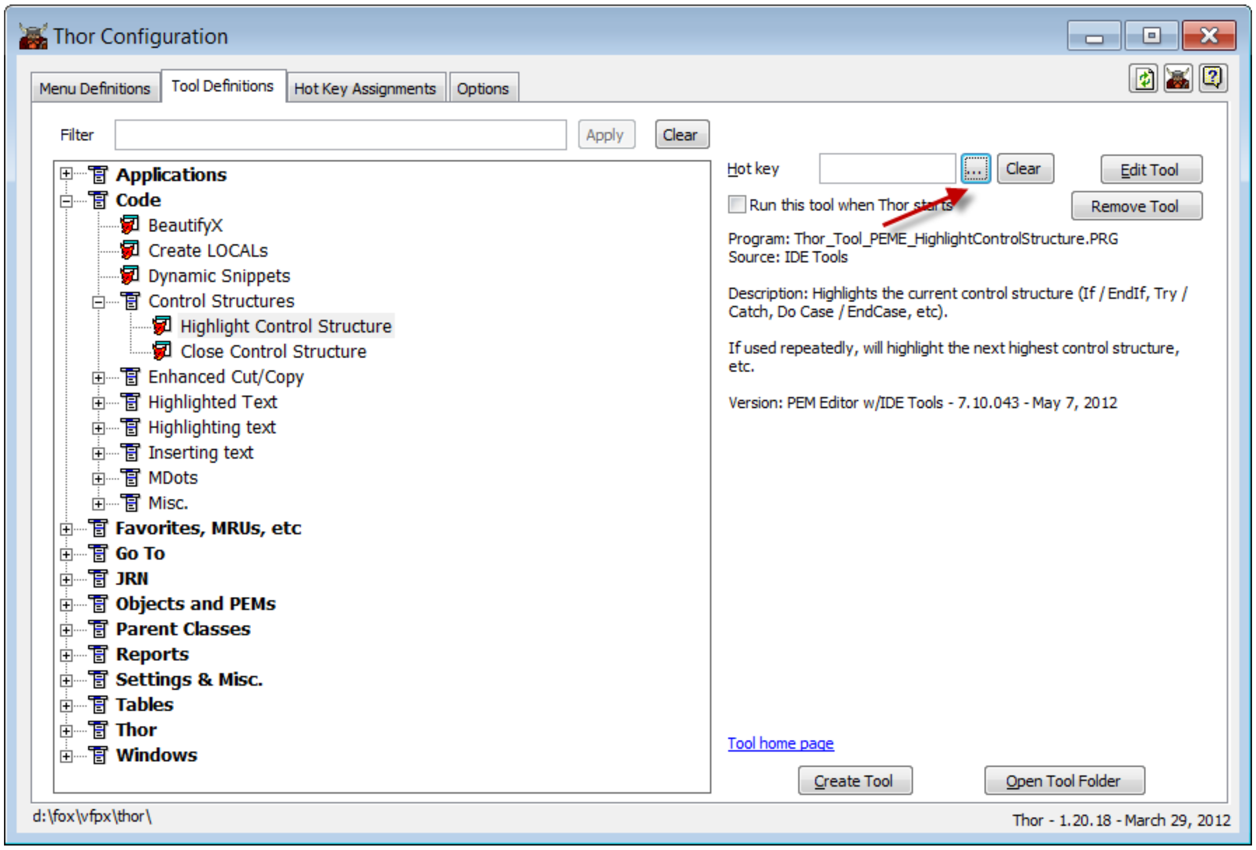
```
SCAN
   * Sum the relevant fields
   m.gtotal = .NULL.
   FOR i = 2 TO FCOUNT() - 1
     IF ISNULL(EVAL(FIELD(m.i)))
         LOOP
     ENDIF
     IF ISNULL(m.gtotal) AND !ISNULL(EVAL(FIELD(m.i)))
         gtotal = 0
     ENDIF
     DO CASE
     CASE THIS.totaltype = COUNT_FIELDS
         * Count values
         IF THIS.shownulls
             gtotal = m.gtotal + IIF(ISNULL(EVAL(FIELD(m.i))),0,1)
         ELSE
             cTmpField = field(m.i)
             gtotal = m.gtotal + IIF(ISBLANK(&cTmpField),0,1)
         ENDIF
     OTHERWISE
         * SUM values
         gtotal = m.gtotal + EVAL(FIELD(m.i))
     ENDCASE
   ENDFOR
   IF THIS.totaltype = PERCENT_FIELDS
         gtotal = IIF(m.sumallflds=0 OR ISNULL(m.gtotal) OR m.gtotal=0,0,ROUND(m.gto
   ENDIF
   REPLACE (m.totfldname) WITH m.gtotal
ENDSCAN
```

Figure 4. The second use of Highlight Control Structure highlights the entire CASE statement.

Of course, if you have to drill down through three layers of menus to use this tool, it probably won't seem all that handy. However, one of the features of Thor is that you can assign a keyboard shortcut to any tool. Before moving on to look at other tools, let's see how you can do so.

Putting tools at your fingertips

Thor offers several mechanisms to make using its tools easier. The simplest is assigning a keystroke combination to a tool, so you can use it without navigating the Thor Tools menu. To do so, open the Thor Configuration form by choosing Thor | Configure from the menu. Click the Tool Definitions tab to open the Tool Definitions page, and navigate in the treeview on the left pane until you find the tool to which you want to add a hotkey. Figure 5 shows the Tool Definitions page with the Highlight Control Structure tool selected.



Program: Thor_Tool_PEME_HighlightControlStructure.PRG
Source: IDE Tools

Description: Highlights the current control structure (If / EndIf, Try / Catch, Do Case / EndCase, etc).

If used repeatedly, will highlight the next highest control structure, etc.

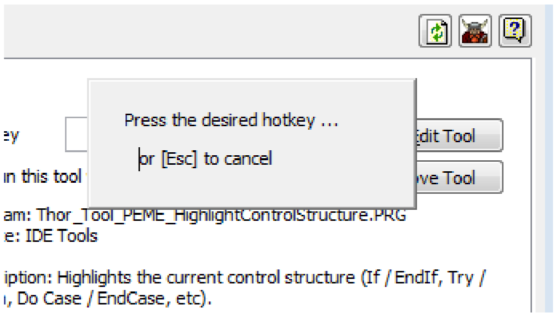
Version: PEM Editor w/IDE Tools - 7.10.043 - May 7, 2012

d:\fox\vfpx\thor\

Thor - 1.20.18 - March 29, 2012

Figure 5. The Tool Definitions page of the Thor Configuration form lets you specify a keyboard combination to run a tool.

Click the ellipsis button (indicated in the figure) and then, as the message that appears (Figure 6) says, press the keyboard combination you want to use. Once you do so, the message disappears, and the textbox shows the specified hotkey. In Figure 7, you can see that I've specified Shift+Ctrl+C as the hot key for Highlight Control Structure.



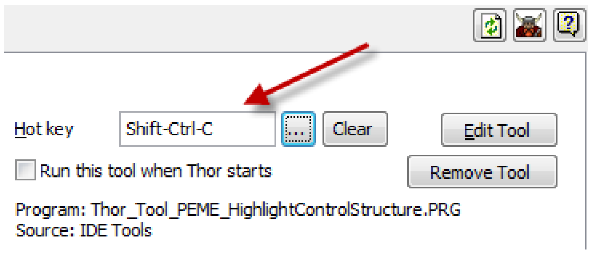
Press the desired hotkey ...

or [Esc] to cancel

am: Thor_Tool_PEME_HighlightControlStructure.PRG
e: IDE Tools

iption: Highlights the current control structure (If / EndIf, Try /
, Do Case / EndCase, etc).

Figure 6. After you click the ellipsis button for a hot key, this message appears. Do as it says to specify a hot key.



Program: Thor_Tool_PEME_HighlightControlStructure.PRG
Source: IDE Tools

Figure 7. Once you type the desired key combination, it appears in the hot key textbox.

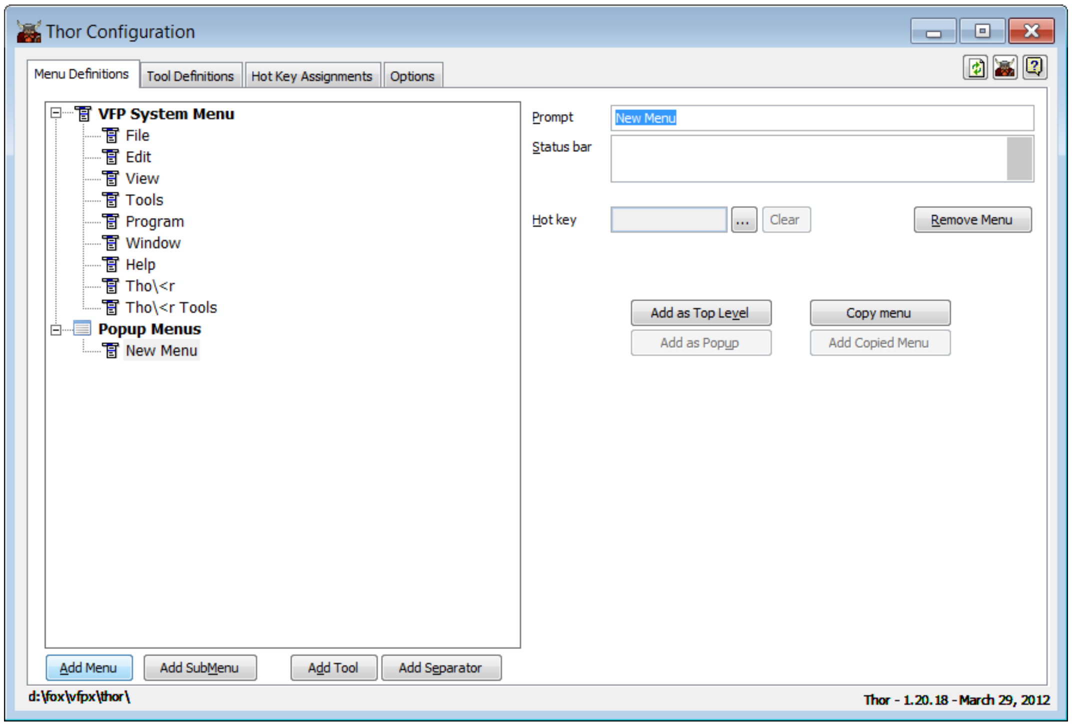
The key combination you specify doesn't take effect until you either close the Thor Configuration form or click the Thor button at the top right of the form to refresh menus and hot keys.

The Thor Configuration form offers a couple of other ways to make individual tools more accessible. You can modify the VFP system menu, adding entire pads, adding submenus to the existing pads or adding tools directly to existing pads. So if you like using the menu, but find that the tools you want to use are buried too deeply in the Thor Tools menu, you can put them where you want them.

In addition, you can create pop-up menus that appear when a specified key combination is pressed. These are like right-click menus, except that they're triggered by the key combination you specify. These pop-up menus can include whichever tools you choose, and can have submenus, if you wish.

To modify the VFP system menu or to create a pop-up menu, use the Menu Definitions page of the Thor Configuration form. Figure 8 shows that page after clicking the Add Menu button with the Popup Menus item highlighted. To define the pop-up menu, specify the prompt (which appears only in the Thor Configuration form) and a hotkey for the pop-up. Then, use the Add Tool button to add one or more tools to the pop-up menu.

In Figure 9, the new pop-up menu has been defined and Thor's two tools that deal with control structures added. Figure 10 shows the newly defined pop-up over a code window.

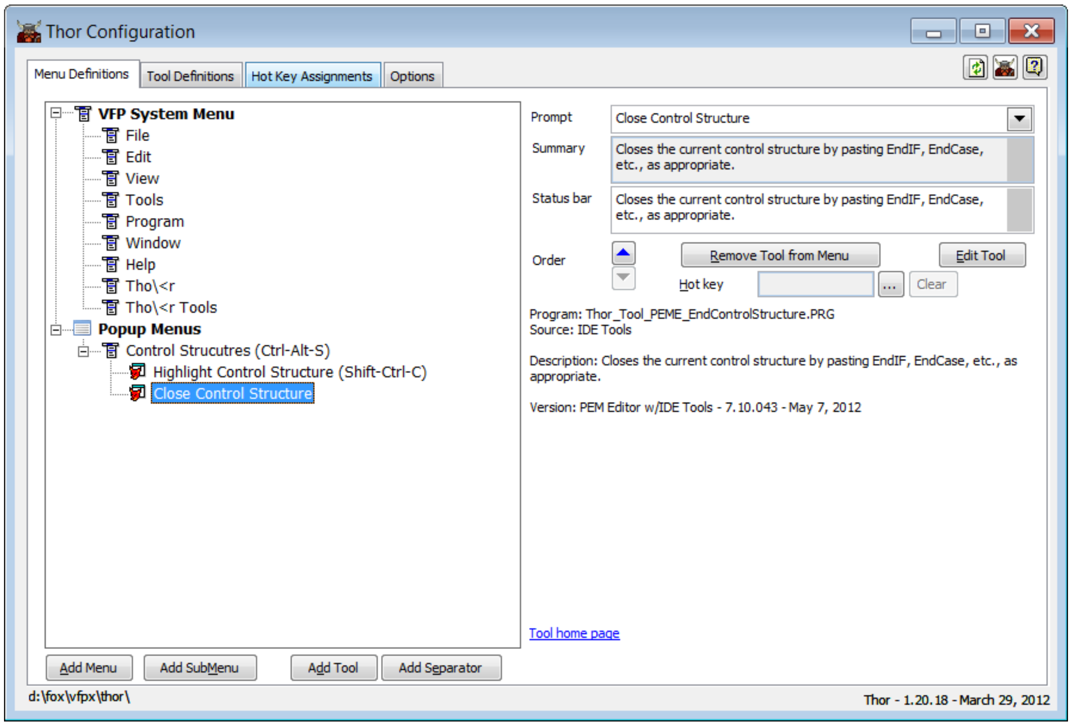


Prompt: New Menu
Status bar:
Hot key:

d:\fox\vfpx\thor\

Thor - 1.20.18 - March 29, 2012

Figure 8. On the Menu Definitions page of the Thor Configuration form, you can add items to the VFP system menu, and create your own pop-up menus.



Prompt: Close Control Structure

Summary: Closes the current control structure by pasting EndIF, EndCase, etc., as appropriate.

Status bar: Closes the current control structure by pasting EndIF, EndCase, etc., as appropriate.

Order

Hot key

Program: Thor_Tool_PEME_EndControlStructure.PRG
Source: IDE Tools

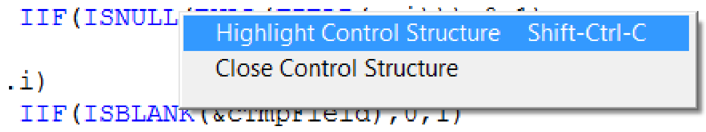
Description: Closes the current control structure by pasting EndIF, EndCase, etc., as appropriate.

Version: PEM Editor w/IDE Tools - 7.10.043 - May 7, 2012

d:\fox\vfpx\thor\

Thor - 1.20.18 - March 29, 2012

Figure 9. A pop-up menu has been defined, containing the two tools related to control structures.



```
IIF(ISNULL
.i)
IIF(ISBLANK
```

Figure 10. When you use the shortcut for the pop-up menu, it appears at the mouse position.

The ability to add hot keys, to modify the VFP system menu, and to define pop-up menus makes it easy for you to decide which Thor Tools you're likely to use and then make those easily accessible.

Edit Parent and Containing Classes

Menu: Parent Classes | Edit Parent and Containing Classes

This tool may well be the one most likely to get people to use Thor. One of VFP's weaknesses is that when you're editing a form or class, you cannot open any class in the inheritance hierarchy of any member of the class being edited. For example, if you're working on a listbox on a form, and you realize that you need to change some code or a setting in the class the listbox is based on, you have to close the form and then open the listbox class. When you're done making changes, you have to close the listbox class and reopen the form.

While Thor can't change that rule, it can make dealing with it easier. That's what this tool is about. It opens a small form showing the classes in the selected object's heritage, and allows you to open any of them (after closing the current class or form). When you do so, the form stays open to allow you to easily get to other listed classes; it also contains a button to reopen the form or class you were originally editing.

For example, the Object Inspector that I built is based on a set of classes that Doug Hennig published. The main form, shown in Figure 11, includes a container class called sfTreeviewExplorer that incorporates a treeview and several other controls. When I run the Edit Parent and Containing Classes tool with that object selected, the form shown in Figure 12 opens. It shows that the control is included in a Form class called sfExplorerFormTreeview and that the control is based on class sfTreeviewExplorer, which inherits from sfTreeviewCursor, which inherits from sfTreeviewContainer, which inherits from sfContainer.

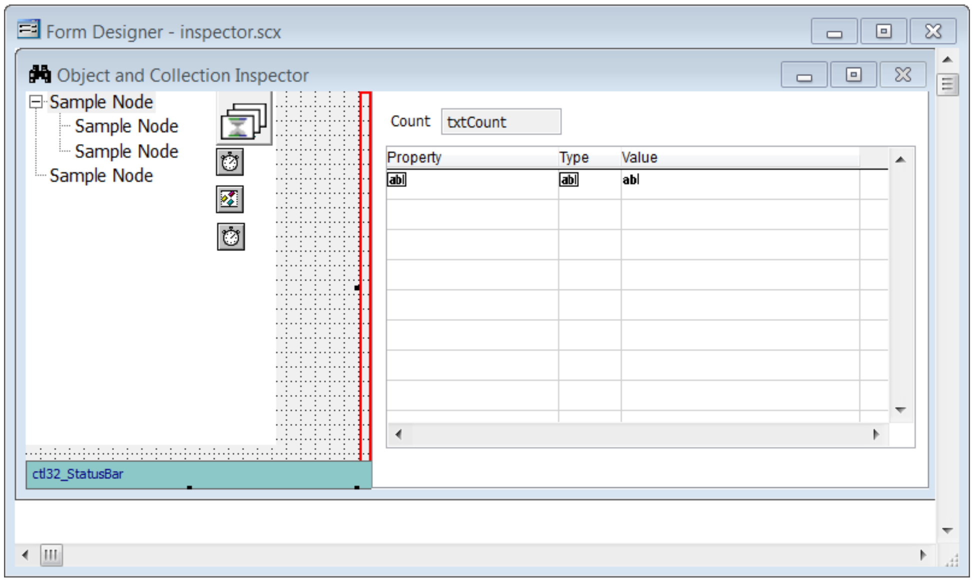
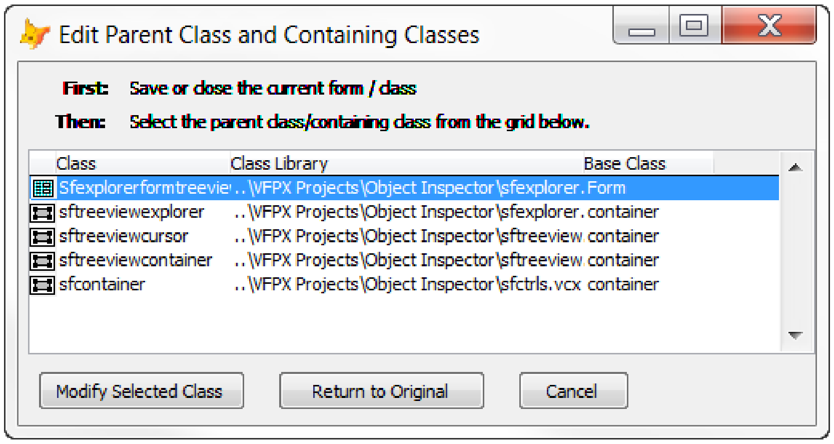


Figure 11. The Object Inspector is based on Doug Hennig's Explorer forms. Here, the sfTreeviewExplorer container object is selected.



Edit Parent Class and Containing Classes

First: Save or close the current form / class

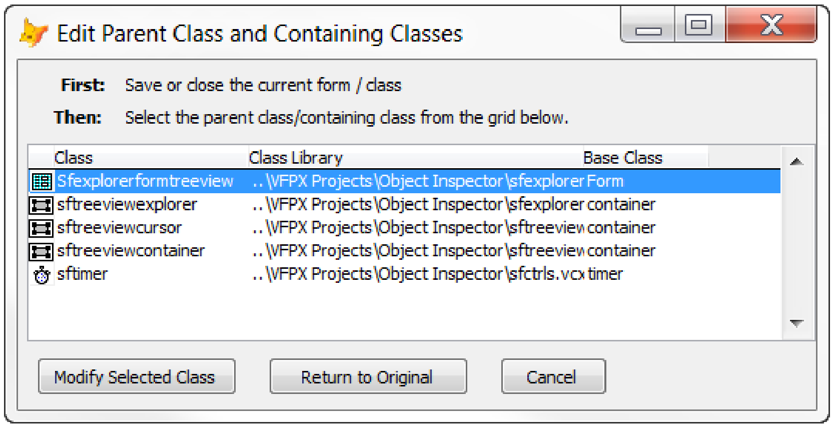
Then: Select the parent class/containing class from the grid below.

Class	Class Library	Base Class
Sfexplorerformtreevie	..\VFPX Projects\Object Inspector\sfexplorer.	Form
sftreeviewexplorer	..\VFPX Projects\Object Inspector\sfexplorer.	container
sftreeviewcursor	..\VFPX Projects\Object Inspector\sftreeview	container
sftreeviewcontainer	..\VFPX Projects\Object Inspector\sftreeview	container
sfcontainer	..\VFPX Projects\Object Inspector\sfctrls.vcx	container

Figure 12. The Edit Parent Class and Containing Classes tool opens this form to let you jump around among the classes in an object's heritage.

The form doesn't actually indicate which classes are in the inheritance hierarchy and which are in the containership hierarchy. Both are included with the containership hierarchy shown first. Figure 13 shows another example from the same form. Before opening the tool this time, the timer control inside the treeview container was selected. The form shows that the timer is contained on a form of class sfExplorerFormTreeview, and also contained in a container of class sfTreeviewExplorer, which inherits from sfTreeviewCursor and sfTreeviewContainer. Finally, the timer is based on class sfTimer.

One thing in Figure 13 might be a little confusing. When looking at the container for the timer, why does the list stop with sfTreeViewContainer? Why doesn't it go all the way back to sfContainer, as in Figure 12. The answer is that the tool doesn't trace the inheritance hierarchy for containing classes. What it does is show you every class in that hierarchy that contains the specified object. So, in this example, sfContainer doesn't include the timer; it was added to sfTreeviewContainer, and then inherited by that class's subclasses, sfTreeviewCursor and sfTreeviewExplorer.



First: Save or close the current form / class

Then: Select the parent class/containing class from the grid below.

Class	Class Library	Base Class
Sfexplorerformtreeview	..\VFPX Projects\Object Inspector\sfexplorer	Form
sftreeviewexplorer	..\VFPX Projects\Object Inspector\sfexplorer	container
sftreeviewcursor	..\VFPX Projects\Object Inspector\sftreeview	container
sftreeviewcontainer	..\VFPX Projects\Object Inspector\sftreeview	container
sftimer	..\VFPX Projects\Object Inspector\sfctrls.vcx	timer

Figure 13. This time, one of the timers inside the sfTreeviewExplorer was selected when this tool was run.

As the instructions at the top of the form indicate, before I can open any of those classes, I have to close the Inspector form. But then, I can choose a class and click Modify Selected Class to open it. Once I've finished editing and closed the Class Designer, I can click Return to Original to reopen the Inspector form exactly as I left it.

Copying and pasting PEMs

While you can copy an object in VFP and paste it onto another form or class, there's no native way to give one object the same property values and method code as another object. A pair of Thor tools provides that capability.

Have you ever set up a control on a form, setting properties and adding code, and then realized that you really want to use a different type of control? For example, you may have used a textbox and realized you want an editbox, or started with a combobox and realized you really want a listbox, or you might want to switch between two classes based on the same base class. Configuring the new control to match the old one is something of a pain, as you have to go one by one through the changed PEMs (properties, events and methods) in the Property Sheet for the original object, and for each, switch to the new object to set its corresponding PEM.

Thor takes the pain away. It also makes it easy to configure a control on one form to match or partially match one on another form, as well as to insert a parent class into an inheritance hierarchy.

Copy (for comparing and pasting)

Menu: Objects and PEMs | Copy / Paste | Copy (for comparing and pasting)

This tool lets you pick up all the modified PEMs for an object and store them on a special clipboard. To use it, you simply select the object and choose this tool.

By itself, this tool isn't terribly useful. It's meant to be followed by Paste properties and method code (described in the next section) or Compare with copied object (described later in this document).

Paste properties and method code

Menu: Objects and PEMs | Copy / Paste | Paste properties and method code

This tool lets you set properties and methods to the values you previously copied from another object. You can choose which PEMs to copy and whether to add properties and methods that don't exist in the target object.

The target object does not have to be based on the same base class as the copied object. That makes this tool great for those situations where you want to switch between two similar classes (though the Re-Define Parent Class tool, described later in this document, actually provides a more direct way to accomplish this task).

Figure 14 shows a form that lets the user select a product (from the Northwind Products table) using a combo. Upon testing, I might find that a listbox provides a better user interface. With this pair of tools, I can change from one to the other in just a few steps.

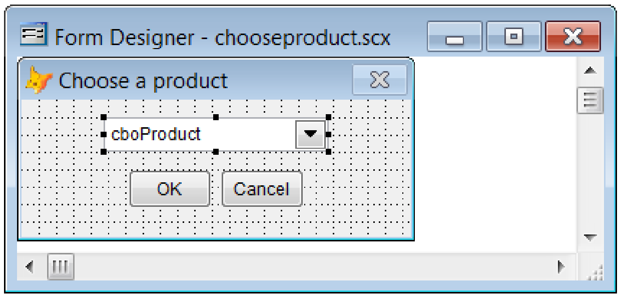
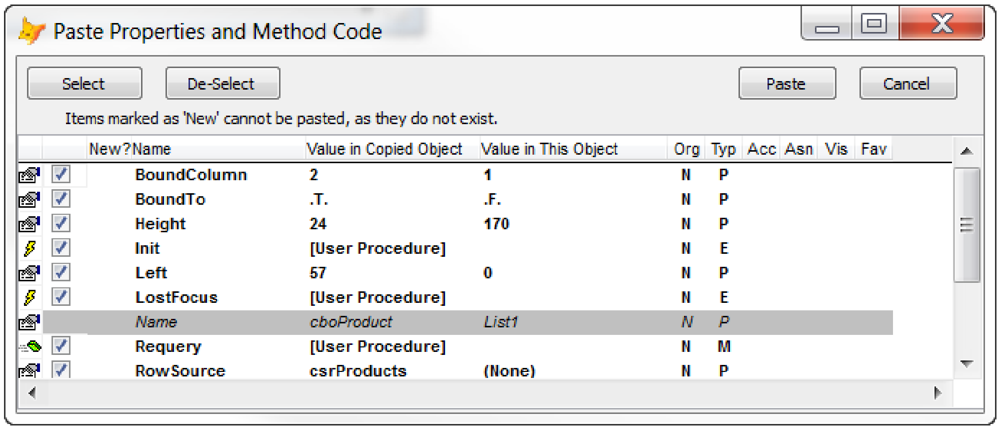


Figure 14. This form uses a combobox to let the user choose a product.

After selecting the combo, choose the Copy (for comparing and pasting) tool to pick up its properties and method code. Then, drop the listbox onto the form (presumably resizing the form). Next, use the Paste properties and method code tool; the form in Figure 15 appears.



Items marked as 'New' cannot be pasted, as they do not exist.

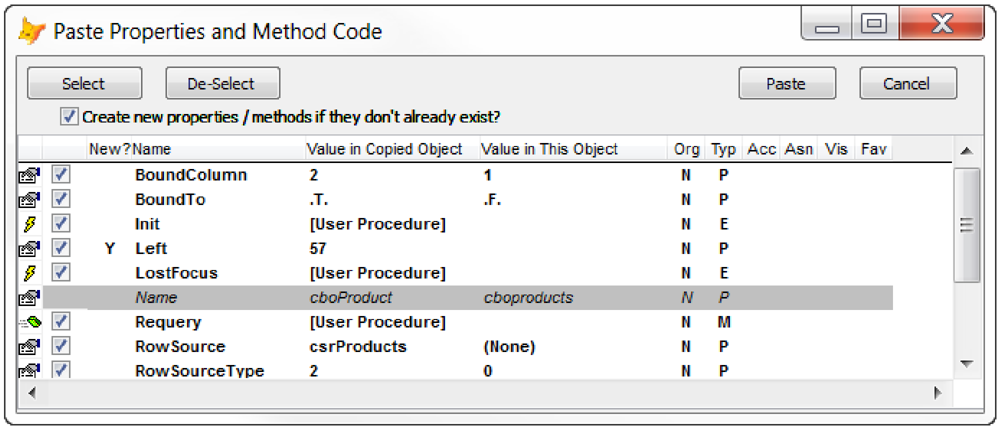
	New?	Name	Value in Copied Object	Value in This Object	Org	Typ	Acc	Asn	Vis	Fav
✓		BoundColumn	2	1	N	P				
✓		BoundTo	.T.	.F.	N	P				
✓		Height	24	170	N	P				
✓		Init	[User Procedure]		N	E				
✓		Left	57	0	N	P				
✓		LostFocus	[User Procedure]		N	E				
		Name	cboProduct	List1	N	P				
✓		Requery	[User Procedure]		N	M				
✓		RowSource	csrProducts	(None)	N	P				

Figure 15. The Paste properties and method code tool displays this form, which allows you to choose which PEMs are copied to the target object.

You can choose which PEMs to copy to the target. In the example, you probably don't want to copy the Height property, since the whole point of using a listbox is to show more products at once. Once you've chosen the ones you want, click the Paste button.

When the target is not a form or a class (that is, the class that's being edited in the Class Designer), properties that exist in the source, but not in the target are marked as New and can't be copied. However, when the target is a form or a class, a checkbox asks whether to create PEMs that don't exist.

Suppose we decide that we'd like to create a combo class for selecting products. After using the Copy (for comparing and pasting) tool, we can create a new combo class and then use Paste properties and method code. The form that appears in this case is shown in Figure 16. (Of course, if you just want to turn a control on a form into a class as is, you can use VFP's native Save As Class option. This tool is better for cases where the class already exists and you want to make a whole set of modifications to match an existing control.)



✓ Create new properties / methods if they don't already exist?

	New?	Name	Value in Copied Object	Value in This Object	Org	Typ	Acc	Asn	Vis	Fav
✓		BoundColumn	2	1	N	P				
✓		BoundTo	.T.	.F.	N	P				
✓		Init	[User Procedure]		N	E				
✓	Y	Left	57		N	P				
✓		LostFocus	[User Procedure]		N	E				
		Name	cboProduct	cboproducts	N	P				
✓		Requery	[User Procedure]		N	M				
✓		RowSource	csrProducts	(None)	N	P				
✓		RowSourceType	2	0	N	P				

Figure 16. When pasting PEMs to a class or form, you can add properties and methods on the fly.

Comparing objects

One of my favorite non-VFP tools is Beyond Compare, which lets me compare folders and files to find differences between them. With Frank Perez's VFP add-on (http://pfsolutions-mi.com/Product/VFP2Text), I can even compare VFP classes and forms. But this isn't a handy way to do so when I'm in the middle of editing them, since you have to close the files to look at them with Beyond Compare.

Thor includes two different tools for comparing objects; both are pretty useful. Compare with Parent Class lets you see which properties of a class have non-default values and shows you the parent class's values for the same properties. Compare with copied object lets you compare two unrelated objects.

Compare with Parent Class

Menu: Parent Classes | Compare with Parent Class

Although the VFP Property Sheet lets you see which PEMs of an object have been changed from their default values (and even lets you see only non-default PEMs), it provides no easy way to see what those values are in the parent class.

The Compare with Parent Class tool opens a separate form that shows each non-default property, event and method in the selected object, along with its value in the selected object and in the parent class. It also indicates those properties with the same value in both places. Finally, it allows you to reset any of the displayed PEMs of the selected object to their default values.

Figure 17 shows the form DataNav.SCX from the Solution Samples that come with VFP. The _tablenav object is selected. Figure 18 shows the form opened by the Compare with Parent Class tool (which is clearly a variation of the one used for the Paste properties and method code tool). It indicates that three properties and five methods have been set in the form's Property Sheet. One of those, Name, is the same on the form as in the class.

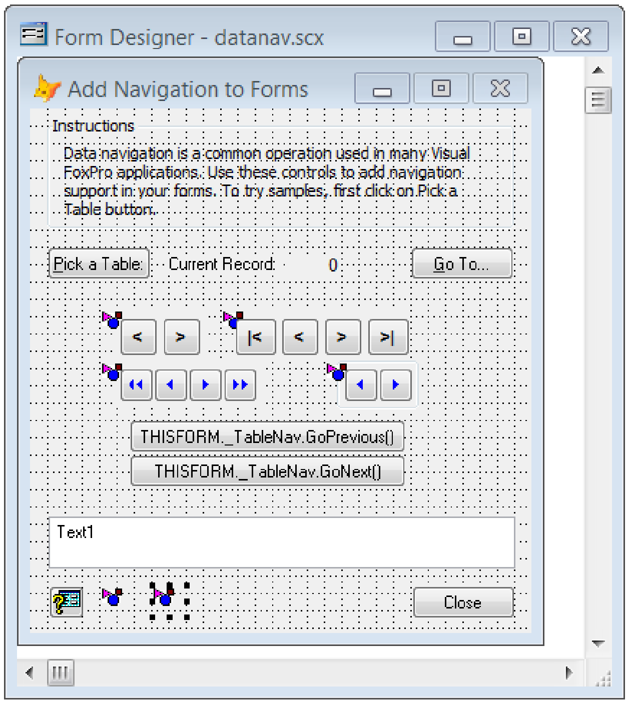
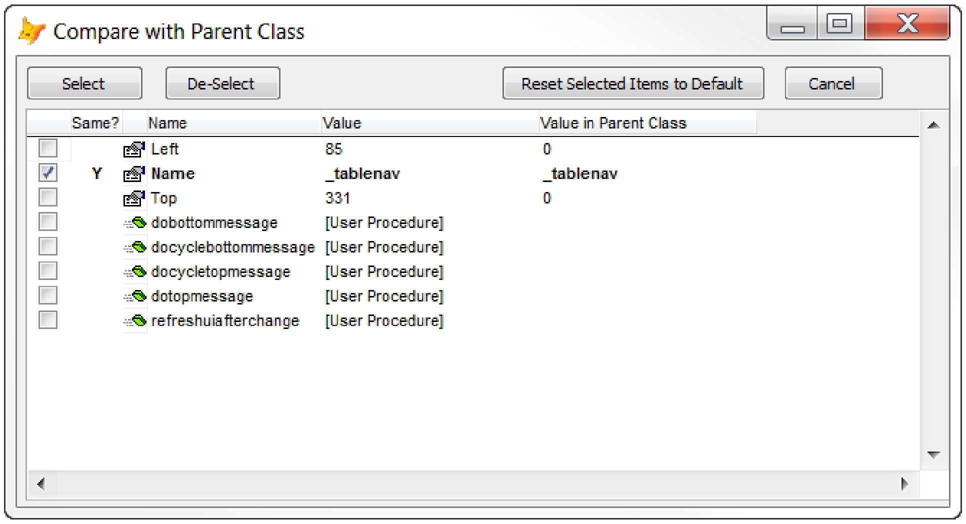

Form Designer - datanav.scx
Add Navigation to Forms
Instructions
Data navigation is a common operation used in many Visual FoxPro applications. Use these controls to add navigation support in your forms. To try samples, first click on Pick a Table button.
Pick a Table
Current Record
0
Go To...
THISFORM._TableNav.GoPrevious()
THISFORM._TableNav.GoNext()
Text1
Close

Figure 17. In this form from the Solutions Samples, an object based on the _tablenav class (from the FoxPro Foundation Classes) is selected.



Same?	Name	Value	Value in Parent Class
☐	Left	85	0
☑ Y	Name	_tablenav	_tablenav
☐	Top	331	0
☐	dobottommessage	[User Procedure]	
☐	docyclebottommessage	[User Procedure]	
☐	docycletopmessage	[User Procedure]	
☐	dotopmessage	[User Procedure]	
☐	refreshuiafterchange	[User Procedure]	

Figure 18. The Compare with Parent Class tool shows only those properties whose values are non-default.

You can use the checkboxes to select some or all of the PEMs shown and then click Reset Selected Items to Default to clear those PEMs in the selected object. Properties that have the same value as in the class are automatically checked.

The Select and De-Select buttons open a dropdown menu (shown in Figure 19) that lets you specify the type of item to check or uncheck.

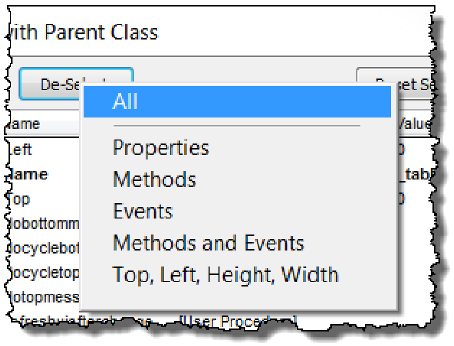


Figure 19. The Select and De-Select buttons of the Compare with Parent Class tool let you choose the type of item to check or uncheck.

There's a reason this tool points out properties that have the same value as in their parent class. Every property that's set in the Property Sheet has to be evaluated and assigned during initialization of a form or other class. So having properties that are explicitly set to the default value from the parent class can slow execution down (though you're unlikely to notice it unless there are many objects with many such properties).

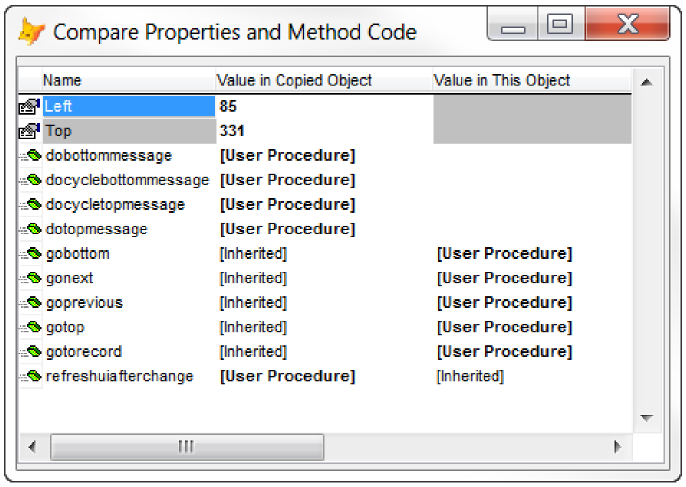
Compare with copied object

Menu: Objects and PEMS | Copy / Paste | Compare with copied object

The second Thor tool for code comparison requires some explanation. It's part of the set of copy-and-paste tools for classes described in "Copying and pasting PEMs," earlier in this document.

To use this tool, you must first select an object and use the Copy (for comparing and pasting) tool found on the same branch of the Thor menu. Then select the object to which you want to compare the first object and choose this tool.

To demonstrate, we'll look at the same object as we did for Compare with Parent Class, even though that tool is a better choice in this case. Starting with the DataNav form, click on the _tablenav object and choose Thor Tools | Objects and PEMs | Copy / Paste | Copy (for comparing and pasting). Now close the form and open the _tablenav class from _table.vcx in the FFC folder (or better yet, use the Edit Parent and Containing Classes tool to open it). When you choose this tool from the menu, the form shown in Figure 20 appears.



Name	Value in Copied Object	Value in This Object
Left	85	
Top	331	
dobottommessage	[User Procedure]	
docyclebottommessage	[User Procedure]	
docycletopmessage	[User Procedure]	
dotopmessage	[User Procedure]	
gobottom	[Inherited]	[User Procedure]
gonext	[Inherited]	[User Procedure]
goprevious	[Inherited]	[User Procedure]
gotop	[Inherited]	[User Procedure]
gotorecord	[Inherited]	[User Procedure]
refreshuiafterchange	[User Procedure]	[Inherited]

Figure 20. This form appears when you use the Compare with copied object tool. It shows PEMs that are different in the two objects.

PEMS with a gray background don't exist in the object where the value column is also gray. In the example, the _tablenav class doesn't have Left and Top properties, but when dropped on the form, they're added. Methods where a column is blank (like the dobottommessage method in the example) exist, but have no code anywhere in the inheritance hierarchy up to that class.

This tool seems particularly useful for situations where one instance of a class isn't working, but other instances are. You can compare the PEMs of a working instance to those of the non-working one to see what you've done differently.

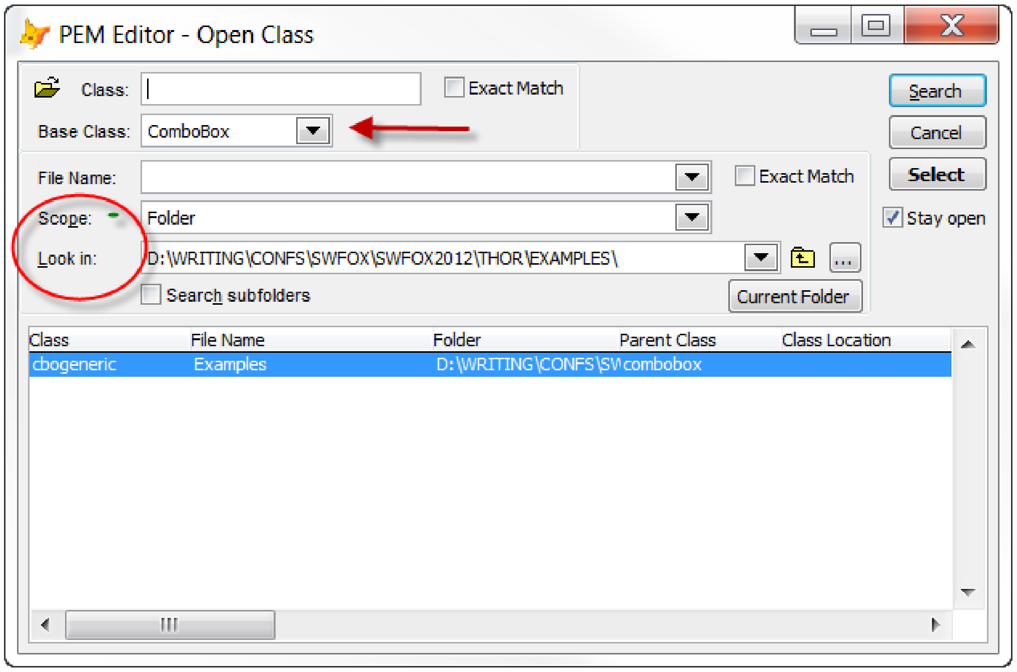
It also seems useful for figuring out whether two existing classes might productively share a common parent class. (When they can, the Paste properties and method code tool gives you an easy way to jump-start creation of the parent class.)

Re-Define Parent Class

Menu: Parent Class | Re-Define Parent Class

Changing the class a control is based on has always been a bit of a pain. The Class Browser lets you do so, and there are various third-party tools like HackCX that handle the task as well. But all of these approaches require you to close the class or form you're working on so that the tool can open it. This tool lets you make the change without closing the form or class.

To use it, select the object whose parent class you want to change, and choose this tool from the menu. The form shown in Figure 21 appears; you use it to find the new parent class. The form lets you specify the type of class you're looking for (indicated by the arrow) and where to look for it. The type can be a specific base class, or "<All>." For scope, you can specify folder (as in the figure) or choose any of the projects on the MRU list. You can also specify all or part of the name of the class you want; unlike the usual VFP string comparison, the portion you specify can be anywhere in the class name. Similarly, you can provide part of a file name to limit the search.



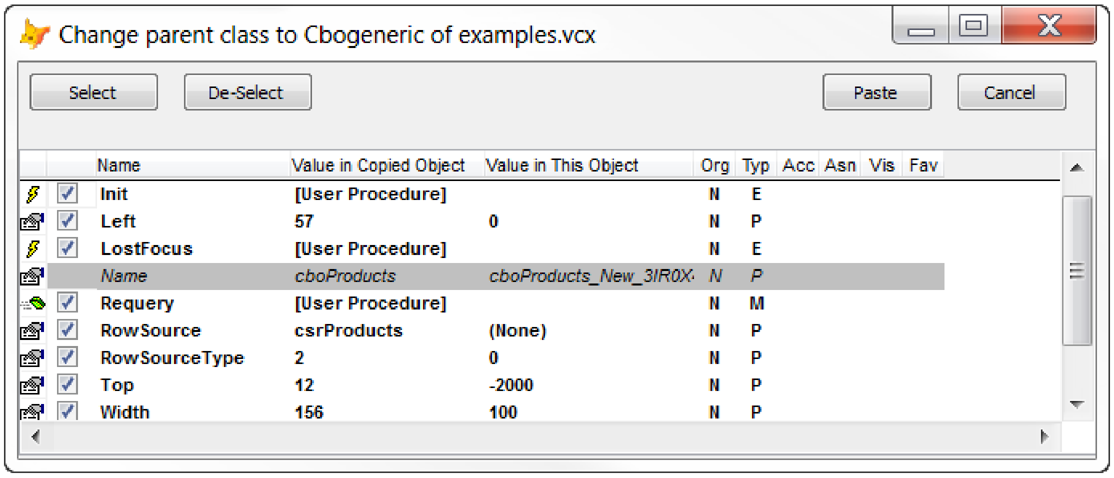
Class:
☐ Exact Match
Base Class: ComboBox
File Name:
☐ Exact Match
Scope: Folder
Look in: D:\WRITING\CONFS\SWFOX\SWFOX2012\THOR\EXAMPLES\
☐ Search subfolders
☑ Stay open

Class	File Name	Folder	Parent Class	Class Location
cbogeneric	Examples	D:\WRITING\CONFS\SW	combobox	

Figure 21. Find and select the new parent class for an object using this form.

When you've set up the type of class and where to look, click the Search button and the grid at the bottom of the form shows all classes that match your specifications. Choose the one you want and click the Select button to change the parent of the selected object to that class.

The same dialog used for the Paste properties and method code and the Compare with parent class tools opens, as in Figure 22, showing you properties and methods that are different in the new parent and the existing object. Decide which of the changed values you want to keep and click Paste.



		Name	Value in Copied Object	Value in This Object	Org	Typ	Acc	Asn	Vis	Fav
	✓	Init	[User Procedure]		N	E				
	✓	Left	57	0	N	P				
	✓	LostFocus	[User Procedure]		N	E				
		Name	cboProducts	cboProducts_New_3IR0X	N	P				
	✓	Requery	[User Procedure]		N	M				
	✓	RowSource	csrProducts	(None)	N	P				
	✓	RowSourceType	2	0	N	P				
	✓	Top	12	-2000	N	P				
	✓	Width	156	100	N	P				

Figure 22. Once you choose the new parent class, you have the opportunity to decide which property values and method code should be kept.

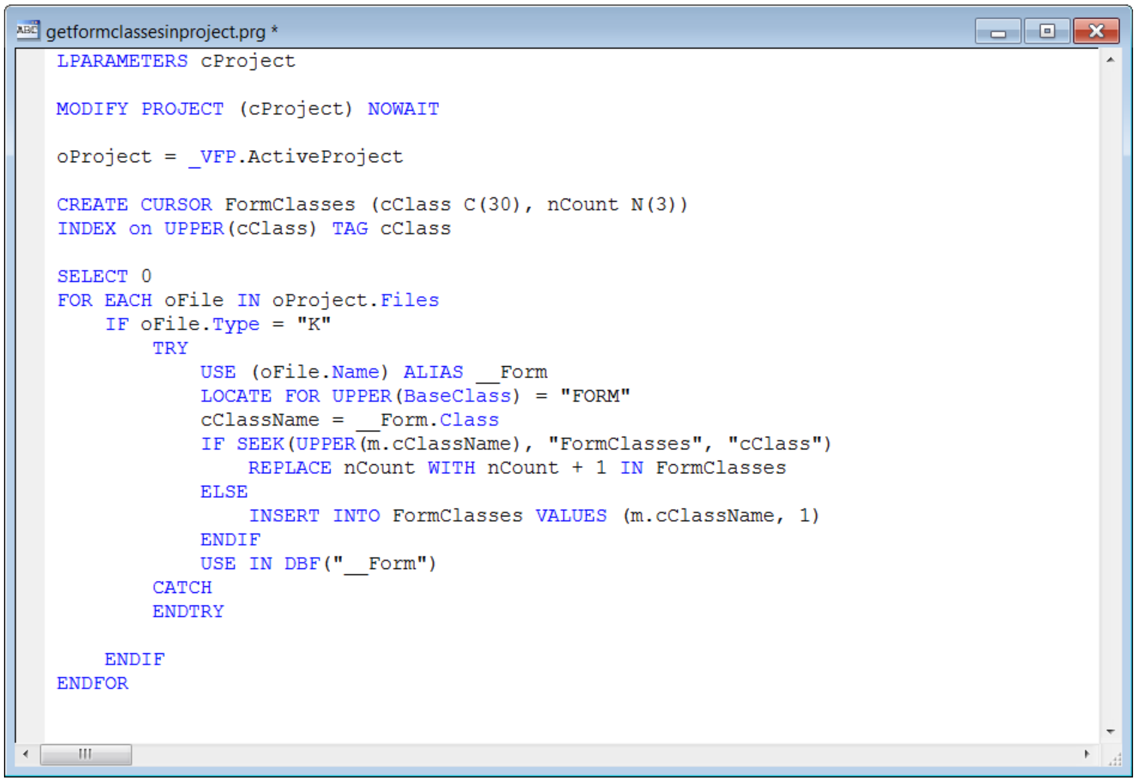
Create Locals

Menu: Code | Create LOCALs

I've always known that declaring all the variables used in a routine is a best practice, and since FoxPro morphed into VFP, that declaring all variables local is the best choice. But the importance of the declarations was really brought home to me by one project. It involved a VFP application that provided a user interface, but also instantiated a VFP COM object. The COM object had a timer, and when the timer fired, the COM object could call methods of the main application's application object. Of course, those calls interrupted whatever was going on in the main application.

While testing this code, we ran into some very weird errors with code working most of the time, but every so often, behaving quite strangely. Eventually, we realized that many of the problems were due to having undeclared variables (which, by default, are private rather than local). The routines called by the timer code used some of the same variable names, and when the variables weren't local, actually changed the values of the variables in the routine that was interrupted by the timer. Once we ensured that every variable in every method was declared local, many of the problems went away.

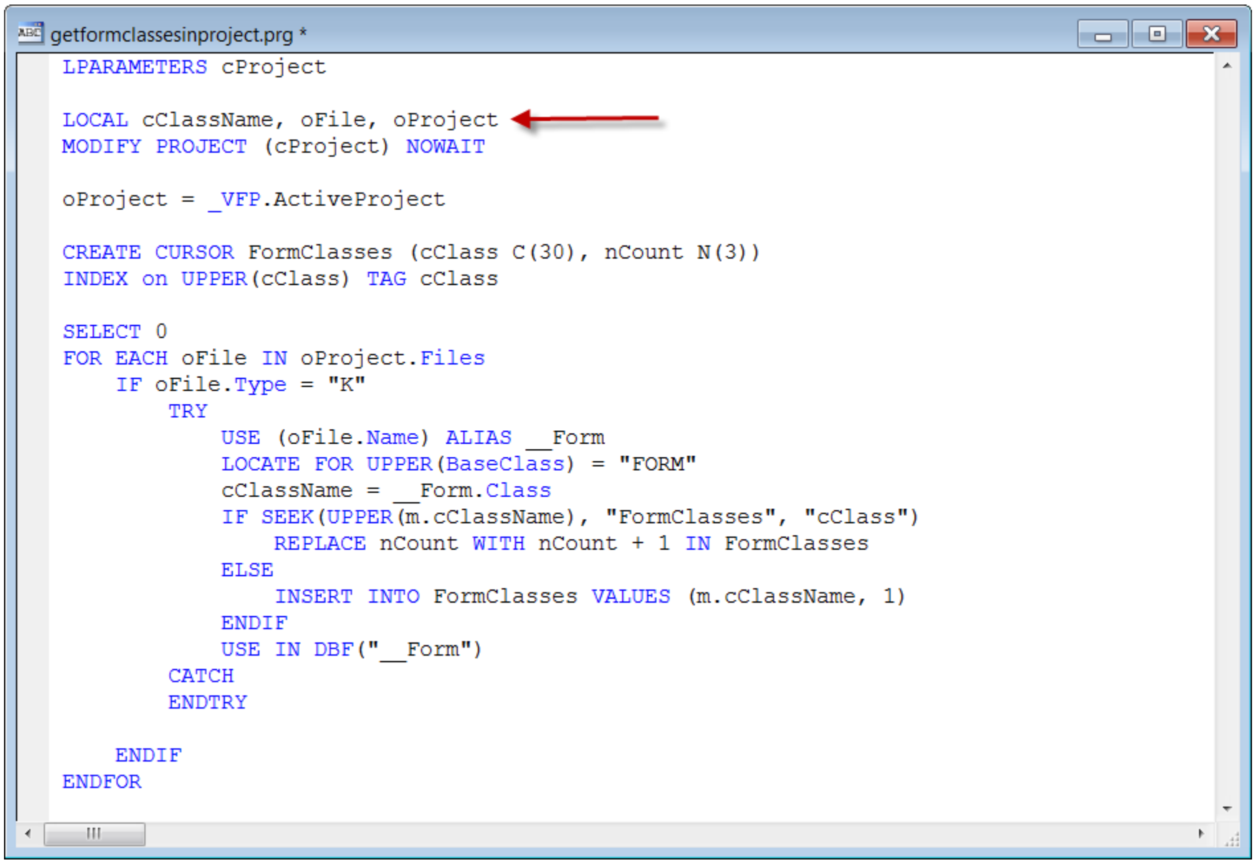
If only I'd had Thor back then. Thor's Create Locals tool takes any code editing window and adds the necessary local declarations. Figure 23 shows a little block of code (that dumps the list of forms in a project into a cursor); it uses several variables, but none of them are declared.



```
LPARAMETERS cProject

MODIFY PROJECT (cProject) NOWAIT

oProject = _VFP.ActiveProject

CREATE CURSOR FormClasses (cClass C(30), nCount N(3))
INDEX on UPPER(cClass) TAG cClass

SELECT 0
FOR EACH oFile IN oProject.Files
    IF oFile.Type = "K"
        TRY
            USE (oFile.Name) ALIAS __Form
            LOCATE FOR UPPER(BaseClass) = "FORM"
            cClassName = __Form.Class
            IF SEEK(UPPER(m.cClassName), "FormClasses", "cClass")
                REPLACE nCount WITH nCount + 1 IN FormClasses
            ELSE
                INSERT INTO FormClasses VALUES (m.cClassName, 1)
            ENDIF
            USE IN DBF("__Form")
        CATCH
        ENDTRY

    ENDIF
ENDFOR
```

Figure 23. This block of code uses several undeclared variables.

Figure 24 shows the same block of code after running the Create Locals tool; the arrow points to the local declarations.



getformclassesinproject.prg *

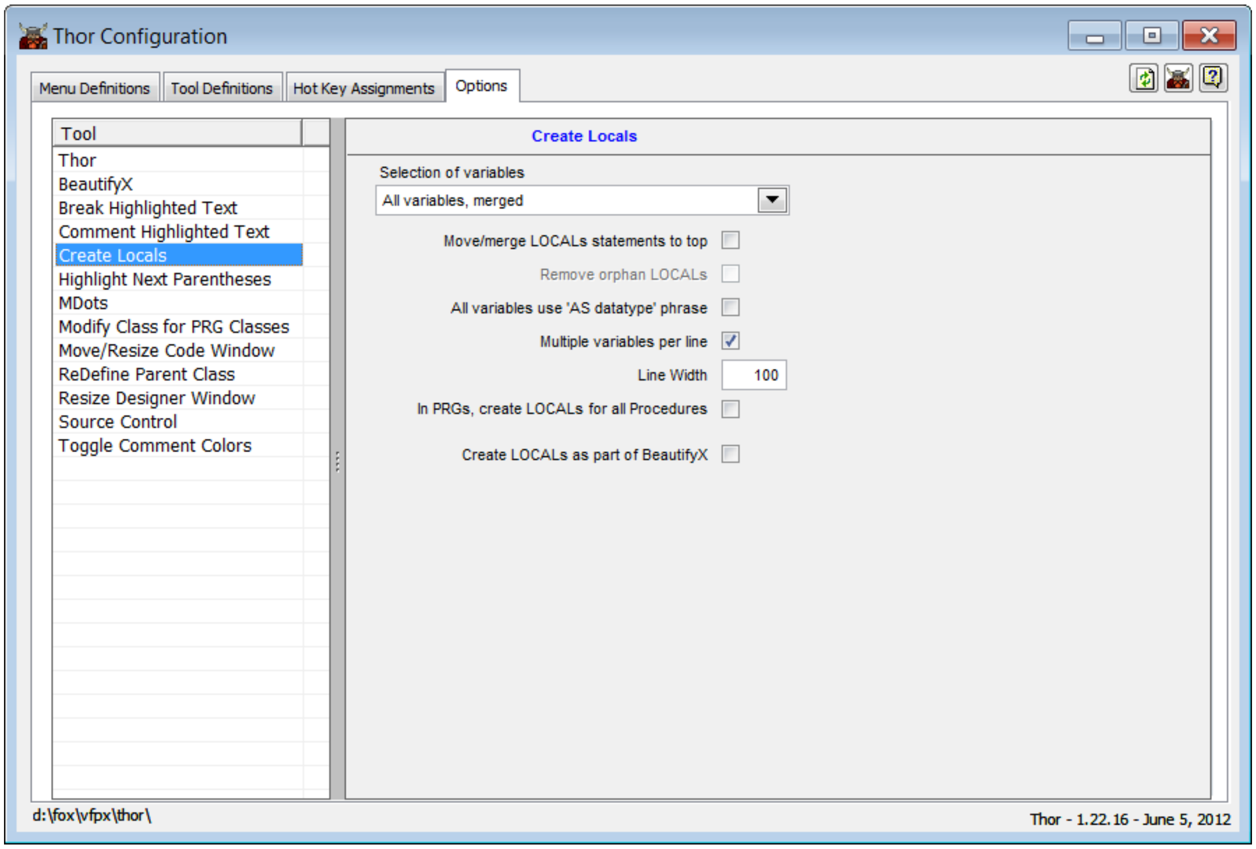
```
LPARAMETERS cProject

LOCAL cClassName, oFile, oProject
MODIFY PROJECT (cProject) NOWAIT

oProject = _VFP.ActiveProject

CREATE CURSOR FormClasses (cClass C(30), nCount N(3))
INDEX on UPPER(cClass) TAG cClass

SELECT 0
FOR EACH oFile IN oProject.Files
    IF oFile.Type = "K"
        TRY
            USE (oFile.Name) ALIAS __Form
            LOCATE FOR UPPER(BaseClass) = "FORM"
            cClassName = __Form.Class
            IF SEEK(UPPER(m.cClassName), "FormClasses", "cClass")
                REPLACE nCount WITH nCount + 1 IN FormClasses
            ELSE
                INSERT INTO FormClasses VALUES (m.cClassName, 1)
            ENDIF
            USE IN DBF("__Form")
        CATCH
        ENDTRY

    ENDIF
ENDFOR
```

Figure 24. After running the Create Locals tool, the same block has variable declarations.

You can control some aspects of this tool's behavior using the Options tab of the Thor Configuration tool.

Figure 25 shows the Options tab with settings for Create Locals displayed, and Figure 26 shows the choices in the Selection of variables dropdown. That dropdown lets you indicate whether all variables should be declared or only those whose names begin with a lowercase "l," the prefix used for local variables in the Hungarian naming convention. Since I use a different naming convention (prefixing variables with their type, but not with "l"), I prefer the "All variables, merged" option.

Most of the other choices should be self-explanatory, or easily understood with a little testing. The last checkbox, Create LOCALs as part of BeautifyX, determines whether local declarations are added when using the BeautifyX tool, which is a replacement for VFP's native Beautify.



Create Locals

Selection of variables: All variables, merged

Move/merge LOCALs statements to top ☐

Remove orphan LOCALs ☐

All variables use 'AS datatype' phrase ☐

Multiple variables per line ✓

Line Width 100

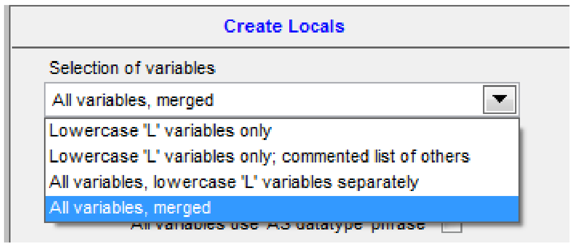
In PRGs, create LOCALs for all Procedures ☐

Create LOCALs as part of BeautifyX ☐

d:\fox\vfpx\thor\

Thor - 1.22.16 - June 5, 2012

Figure 25. This page lets you determine how the Create Locals tool behaves.



Create Locals

Selection of variables

All variables, merged

- Lowercase 'L' variables only
- Lowercase 'L' variables only; commented list of others
- All variables, lowercase 'L' variables separately
- All variables, merged

Figure 26. Create Locals can apply to only a subset of variables, based on their names.

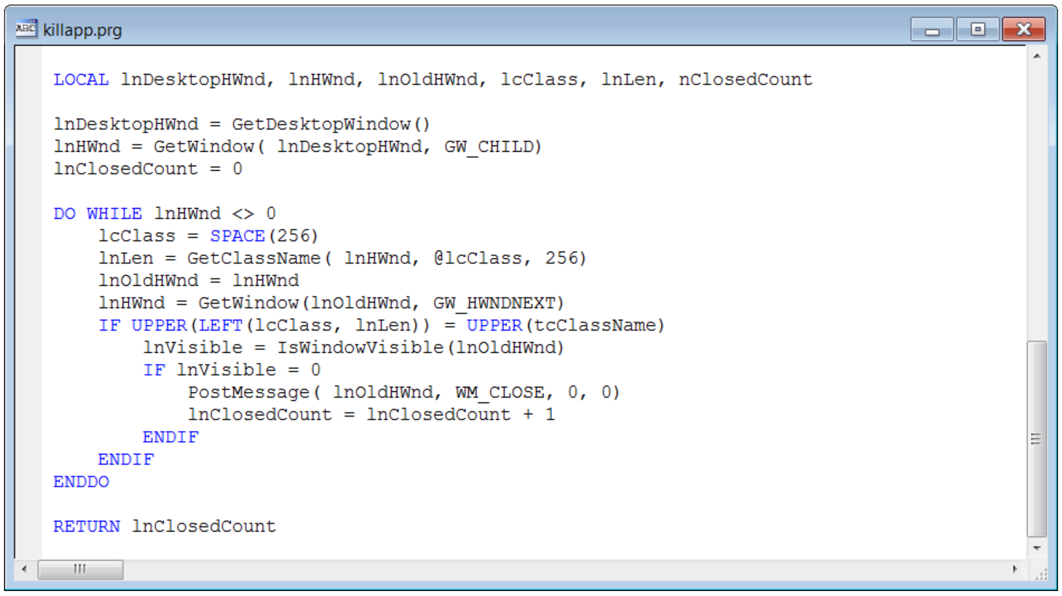
If, even after trying different settings, you don't like the way the local declarations are created, Thor offers you the ultimate flexibility. You can create your own version of the code for the tool. To do so, choose More | Manage Plug-Ins from the Thor menu. In the form that appears, find CreateLocalsStatements and click the Create button next to it. That opens a program containing the current code use to actually create the local declarations. You can modify it and save it (it's automatically in the right place), and from then on, the tool will use your modified version.

Add MDots to variable names

Menu: Code | MDots | Add MDots to variable names

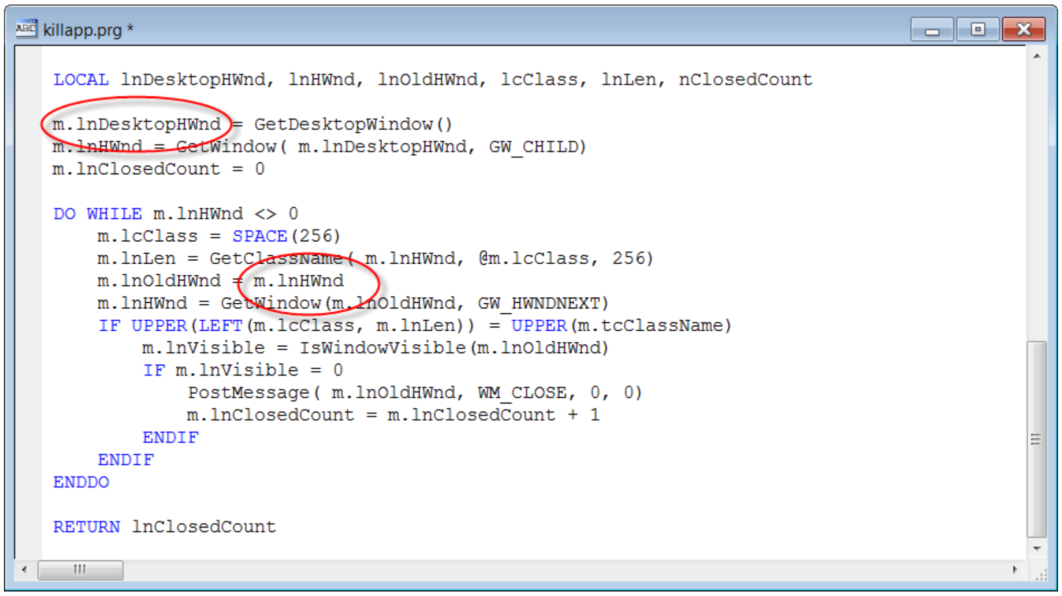
As with declaring all variables local, I've known forever that all accesses of a variable should be preceded by "m." (often written as "MDot") to ensure that VFP looks at the variable rather than at a field of the same name. In fact, increasingly, I remember to put them in my code, but I still forget sometimes. This Thor tool catches all the places I missed.

Figure 27 shows a block of code that doesn't use the MDot notation. Figure 28 shows the same code after using this tool. A couple of the changes have been circled.



```
LOCAL lnDesktopHWnd, lnHWnd, lnOldHWnd, lcClass, lnLen, nClosedCount

lnDesktopHWnd = GetDesktopWindow()
lnHWnd = GetWindow( lnDesktopHWnd, GW_CHILD)
lnClosedCount = 0

DO WHILE lnHWnd <> 0
    lcClass = SPACE(256)
    lnLen = GetClassName( lnHWnd, @lcClass, 256)
    lnOldHWnd = lnHWnd
    lnHWnd = GetWindow(lnOldHWnd, GW_HWNDNEXT)
    IF UPPER(LEFT(lcClass, lnLen)) = UPPER(tcClassName)
        lnVisible = IsWindowVisible(lnOldHWnd)
        IF lnVisible = 0
            PostMessage( lnOldHWnd, WM_CLOSE, 0, 0)
            lnClosedCount = lnClosedCount + 1
        ENDIF
    ENDIF
ENDDO

RETURN lnClosedCount
```

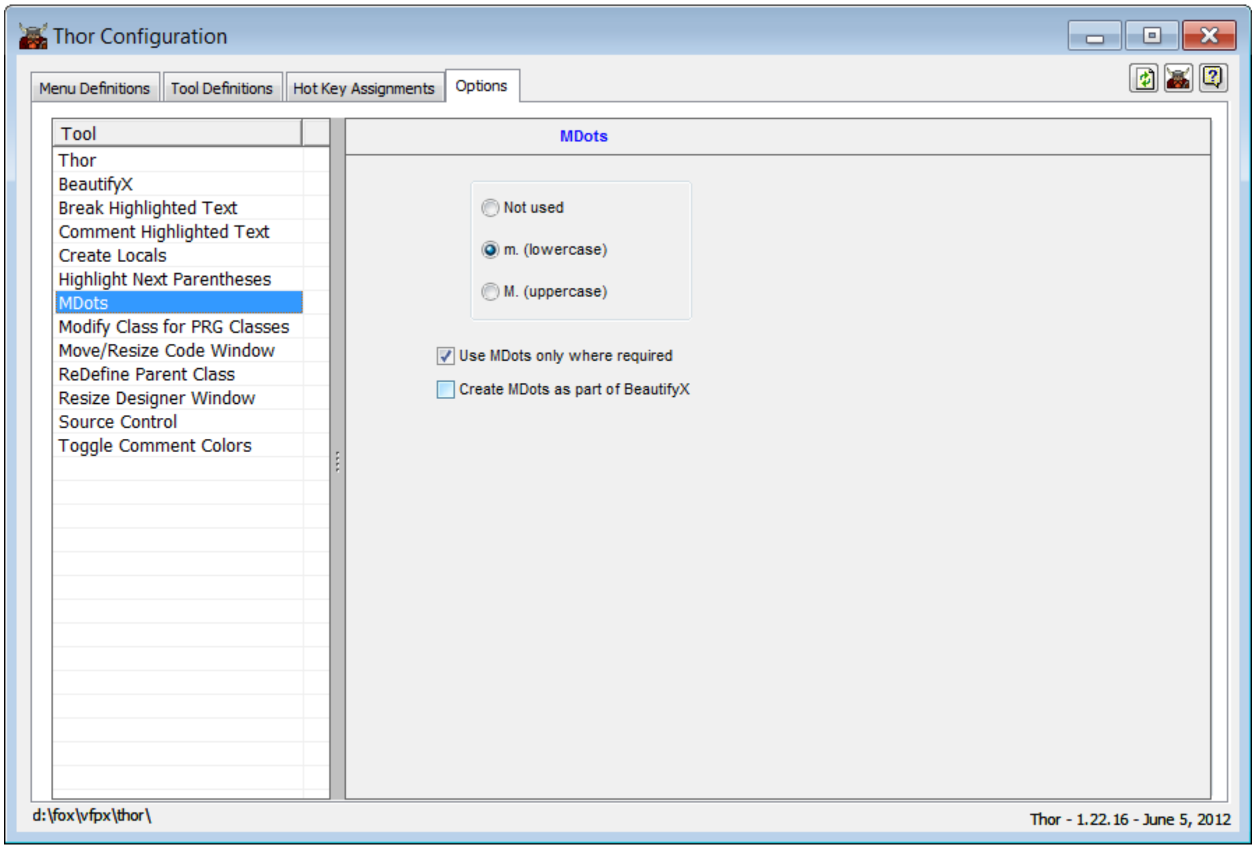
Figure 27. This code doesn't use the MDOT notation to ensure no conflicts between variables and field names.



```
LOCAL lnDesktopHWnd, lnHWnd, lnOldHWnd, lcClass, lnLen, nClosedCount

m.lnDesktopHWnd = GetDesktopWindow()
m.lnHWnd = GetWindow( m.lnDesktopHWnd, GW_CHILD)
m.lnClosedCount = 0

DO WHILE m.lnHWnd <> 0
    m.lcClass = SPACE(256)
    m.lnLen = GetClassName( m.lnHWnd, @m.lcClass, 256)
    m.lnOldHWnd = m.lnHWnd
    m.lnHWnd = GetWindow(m.lnOldHWnd, GW_HWNDNEXT)
    IF UPPER(LEFT(m.lcClass, m.lnLen)) = UPPER(m.tcClassName)
        m.lnVisible = IsWindowVisible(m.lnOldHWnd)
        IF m.lnVisible = 0
            PostMessage( m.lnOldHWnd, WM_CLOSE, 0, 0)
            m.lnClosedCount = m.lnClosedCount + 1
        ENDIF
    ENDIF
ENDDO

RETURN lnClosedCount
```

Figure 28. After using the Add MDot to variable names tool, the code from Figure 27 has “m.” before all references to variables.

By default, this tool adds MDots a few places where they’re not necessary, such as on the left-hand side of assignment statement. However, you can control this behavior using the Options tab of the Thor Configuration form, shown in Figure 29. Figure 30 shows the same block of code when using the tool as configured in Figure 29.



Tool
Thor
BeautifyX
Break Highlighted Text
Comment Highlighted Text
Create Locals
Highlight Next Parentheses
MDots
Modify Class for PRG Classes
Move/Resize Code Window
ReDefine Parent Class
Resize Designer Window
Source Control
Toggle Comment Colors

MDots

Not used
[✓ m. (lowercase)]
M. (uppercase)

[✓] Use MDots only where required
[ ] Create MDots as part of BeautifyX

d:\fox\vfpx\thor\
Thor - 1.22.16 - June 5, 2012

Figure 29. Thor offers options for various tools. Here, you can determine whether the Add MDots tool uses lower-case or upper-case and whether it puts MDots in front of all uses of variables or only those where it's required to avoid conflict with field names.

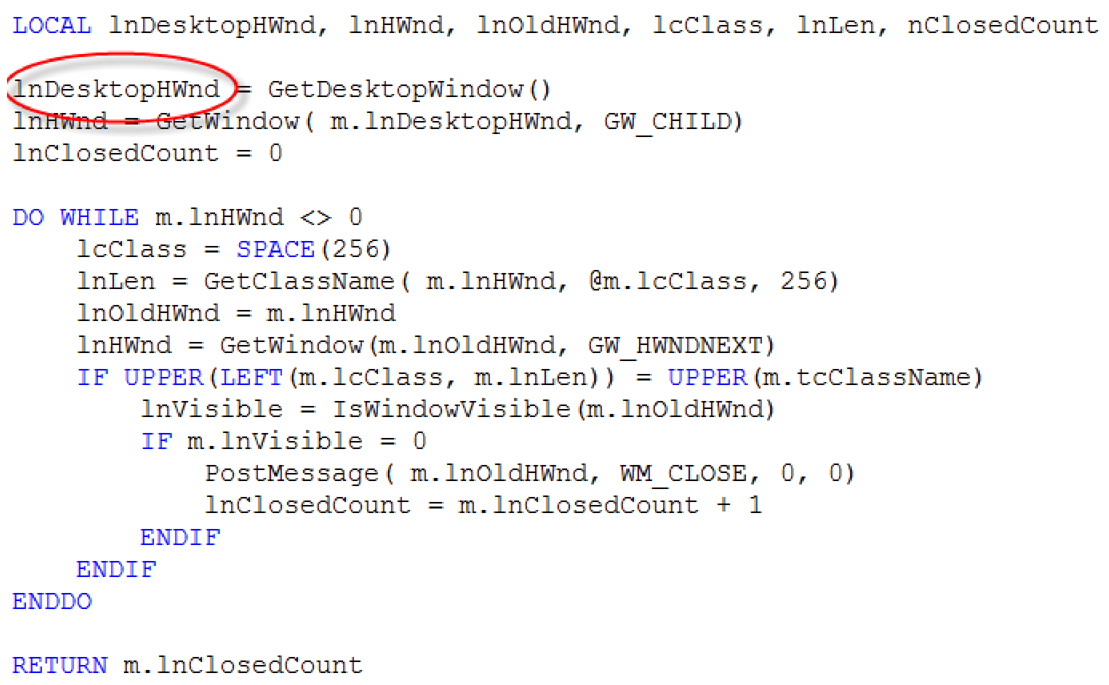

```
LOCAL lnDesktopHWnd, lnHWnd, lnOldHWnd, lcClass, lnLen, nClosedCount

lnDesktopHWnd = GetDesktopWindow()
lnHWnd = GetWindow( m.lnDesktopHWnd, GW_CHILD)
lnClosedCount = 0

DO WHILE m.lnHWnd <> 0
    lcClass = SPACE(256)
    lnLen = GetClassName( m.lnHWnd, @m.lcClass, 256)
    lnOldHWnd = m.lnHWnd
    lnHWnd = GetWindow(m.lnOldHWnd, GW_HWNDNEXT)
    IF UPPER(LEFT(m.lcClass, m.lnLen)) = UPPER(m.tcClassName)
        lnVisible = IsWindowVisible(m.lnOldHWnd)
        IF m.lnVisible = 0
            PostMessage( m.lnOldHWnd, WM_CLOSE, 0, 0)
            lnClosedCount = m.lnClosedCount + 1
        ENDIF
    ENDIF
ENDDO

RETURN m.lnClosedCount
```

Figure 30. After changing the option to include MDots only where required, variables on the left-hand side of an assignment statement no longer get the MDot prefix.

Highlight Parentheses

Menu: Code | Highlighting text | Highlight parentheses

This is another tool that's especially handy when I'm exploring code written by someone else or old code I haven't seen in a while. It's also great for those times when you're getting a syntax error and can't see what's wrong. When you run this tool, it looks both ways from the cursor position to find a matching pair of parentheses and highlights the matching parentheses and all the code in between. Figure 31 shows a line of code with nested parentheses; the cursor is inside the inner block. Figure 32 shows the same block after using the tool.

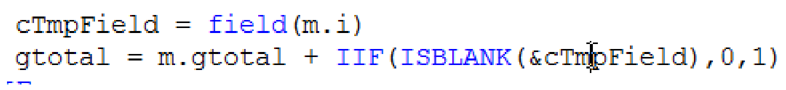

```
cTmpField = field(m.i)
gtotal = m.gtotal + IIF(ISBLANK(&cTmpField),0,1)
```

Figure 31. The cursor here is inside parentheses.

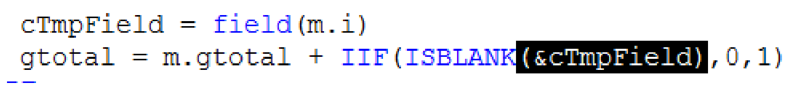
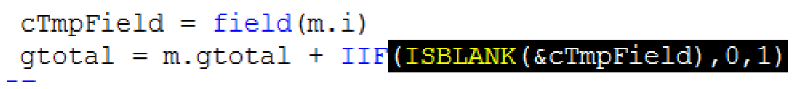

```
cTmpField = field(m.i)
gtotal = m.gtotal + IIF(ISBLANK(&cTmpField),0,1)
```

Figure 32. Highlight parentheses finds the first containing pair of parentheses and highlights the contained code.

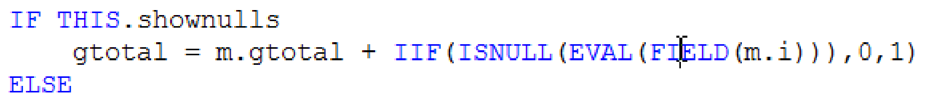
Like Highlight Control Structure, using this tool repeatedly moves outward; Figure 33 shows the same block of code after using the tool twice. Oddly, if there are no more pairs of parentheses containing the highlighted code, the tool highlights all the code in the editing window.



```
cTmpField = field(m.i)
gtotal = m.gtotal + IIF(ISBLANK(&cTmpField),0,1)
```

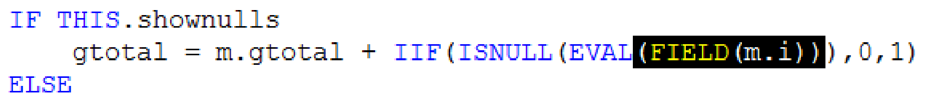
Figure 33. Each subsequent use of Highlight parentheses moves out by one pair of parentheses.

Highlight parentheses is smart enough to get things right when the matching pair of parentheses contains other parentheses. For example, in Figure 34, the cursor is positioned on the FIELD() function, but not inside its parentheses. Figure 35 shows the result of using the tool. The correct pairing is found.



```
IF THIS.shownulls
    gtotal = m.gtotal + IIF(ISNULL(EVAL(FIELD(m.i))),0,1)
ELSE
```

Figure 34. Here, the cursor's initial position isn't in the innermost pair of parentheses.



```
IF THIS.shownulls
    gtotal = m.gtotal + IIF(ISNULL(EVAL(FIELD(m.i))),0,1)
ELSE
```

Figure 35. The Highlight parentheses tool gets it right, even when the initial cursor position isn't inside the innermost parentheses.

Document Treeview

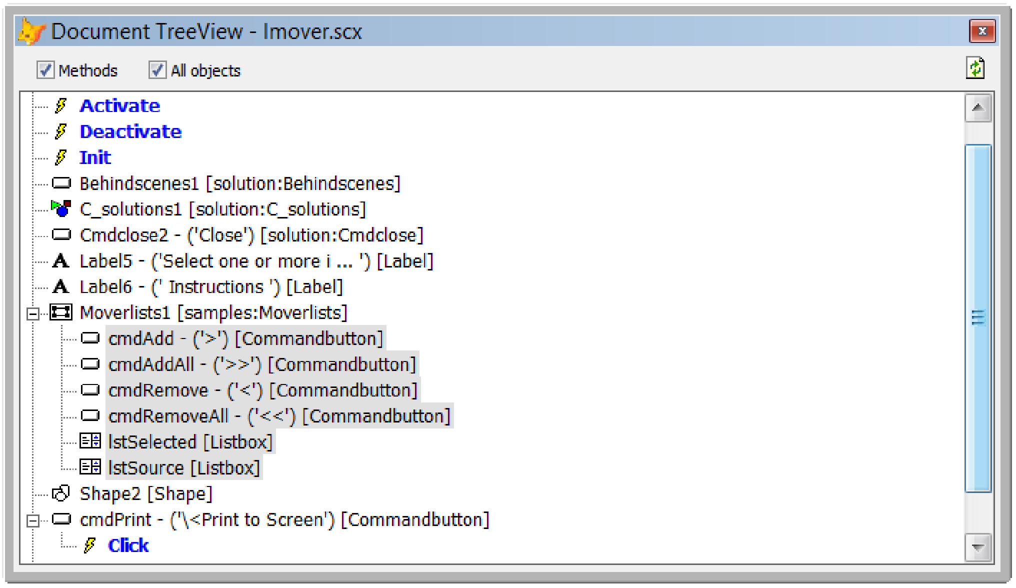
Menu: Applications | Document Treeview

When Document View was added in VFP 7, it gave us an easy way to navigate inside files containing multiple routines. It was a major improvement over the Procedures and Functions List it replaced. I use it all the time when working with classes created in code (PRG).

But Document View has never been all that helpful for forms or for classes stored in a class library (VCX). While it lists every method that contains code, it doesn't give any sense of structure, and in a busy form or class, it can be quite cluttered.

The Document Treeview tool is better suited for visual classes (and, in fact, doesn't work for code). Document Treeview is based on the main combobox of PEM Editor, but you can use it as a stand-alone tool. It shows the objects in a form or class and those of their methods that contain code. As the name indicates, it uses a treeview control to organize the information, so you can expand or collapse any section. It can be resized as well as docked. In addition, you can control what appears at any time.

Figure 36 shows Document Treeview for the lMover form from the Solution Samples (found in folder Samples\Solution\Controls\Lists\); some objects and methods are out-of-sight above and below the section shown.



Document TreeView - lmover.scx

☑ Methods ☑ All objects

```
Activate
Deactivate
Init
Behindscenes1 [solution:Behindscenes]
C_solutions1 [solution:C_solutions]
Cmdclose2 - ('Close') [solution:Cmdclose]
Label5 - ('Select one or more i ... ') [Label]
Label6 - (' Instructions ') [Label]
Moverlists1 [samples:Moverlists]
    cmdAdd - ('>') [Commandbutton]
    cmdAddAll - ('>>') [Commandbutton]
    cmdRemove - ('<') [Commandbutton]
    cmdRemoveAll - ('<<') [Commandbutton]
    lstSelected [Listbox]
    lstSource [Listbox]
Shape2 [Shape]
cmdPrint - ('\<Print to Screen') [Commandbutton]
    Click
```

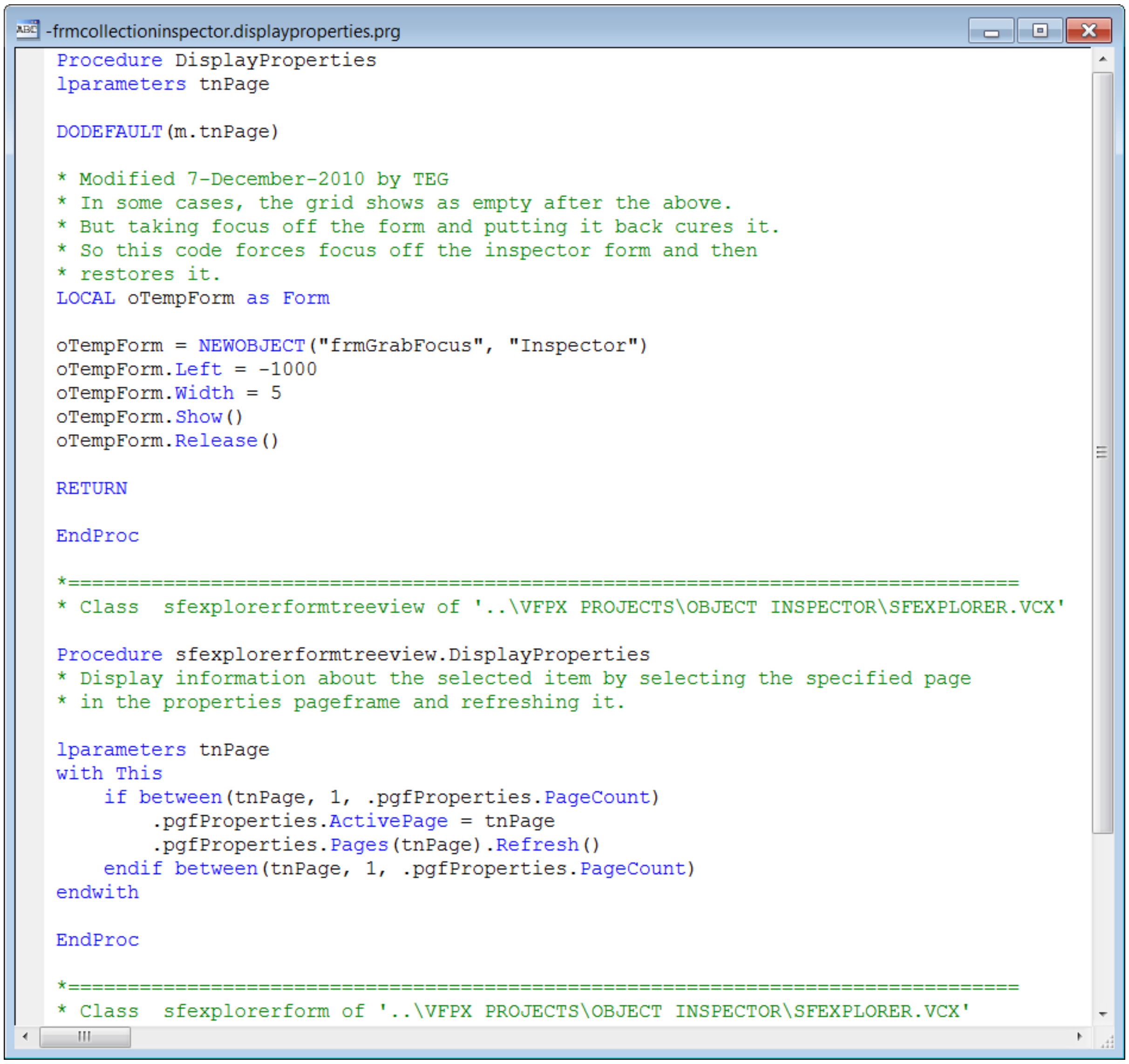
Figure 36. Document Treeview shows you the objects in a class or form and any methods that contain code.

The colors in Document Treeview help to quickly understand the display. Names of objects are shown in black. If the object was inherited along with its container from a parent class (like the buttons inside Moverlists1 in the example), its backcolor is gray. Method names are shown in blue if they contain code at this level; if they have only inherited code, they're shown in gray. All method names are bold, which offers another way to distinguish them from object names.

Clicking on an object makes it the current object in PEM Editor, if that tool is open. Often, it also selects that object in the Form or Class Designer and makes it the current object in the Property Sheet; there are cases where it's not possible to do so programmatically, however.

You can click on any method to open it for editing; that's just like Document View. But Document Treeview offers another possibility. Use Ctrl+Click on the method and a window opens showing you all code for that method at all levels of the inheritance hierarchy. Of course, you can't edit the code there, but it's a much easier way to see all the code than anything offered natively. (Another Thor Tool, Code Listings, also lets you see all the code in one place.)

Figure 37 shows the window that opens when you Ctrl+Click on the DisplayProperties method of the main form for my Object Inspector tool. The code added at the form level is shown first, followed by the code inherited from the sfExplorerFormTreeview class, followed by the code for this method in sfExplorerForm, the level at which the method was defined.

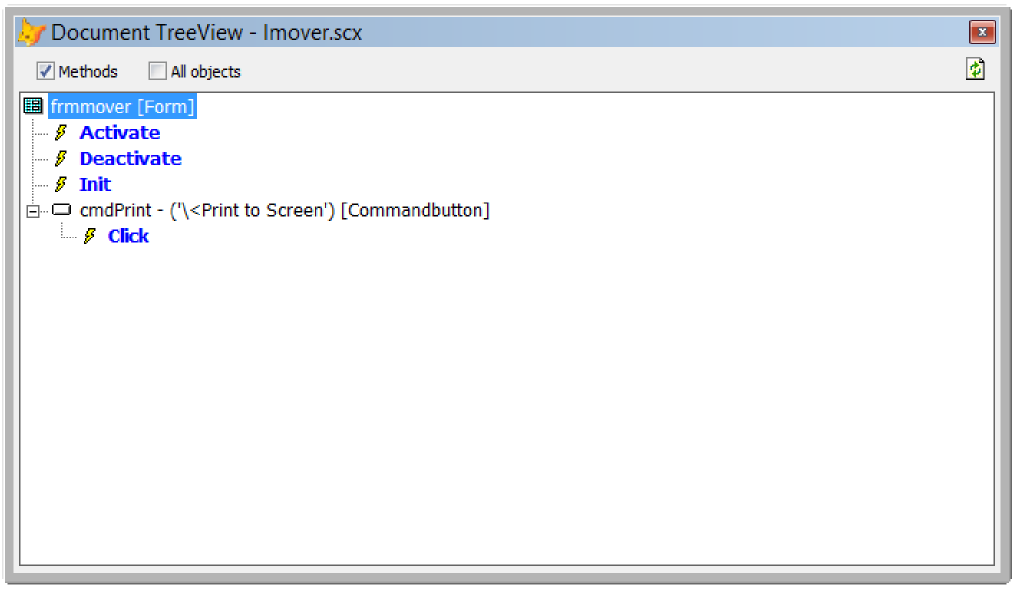


-frmcollectioninspector.displayproperties.prg

```
Procedure DisplayProperties
lparameters tnPage

DODEFAULT(m.tnPage)

* Modified 7-December-2010 by TEG
* In some cases, the grid shows as empty after the above.
* But taking focus off the form and putting it back cures it.
* So this code forces focus off the inspector form and then
* restores it.
LOCAL oTempForm as Form

oTempForm = NEWOBJECT("frmGrabFocus", "Inspector")
oTempForm.Left = -1000
oTempForm.Width = 5
oTempForm.Show()
oTempForm.Release()

RETURN

EndProc

*==================================================================================
* Class  sfexplorerformtreeview of '..\VFPX PROJECTS\OBJECT INSPECTOR\SFEXPLORER.VCX'

Procedure sfexplorerformtreeview.DisplayProperties
* Display information about the selected item by selecting the specified page
* in the properties pageframe and refreshing it.

lparameters tnPage
with This
    if between(tnPage, 1, .pgfProperties.PageCount)
        .pgfProperties.ActivePage = tnPage
        .pgfProperties.Pages(tnPage).Refresh()
    endif between(tnPage, 1, .pgfProperties.PageCount)
endwith

EndProc

*==================================================================================
* Class  sfexplorerform of '..\VFPX PROJECTS\OBJECT INSPECTOR\SFEXPLORER.VCX'
```

Figure 37. Using Ctrl+Click on a method in Method Treeview shows you all code for that method, from all levels of the inheritance hierarchy.

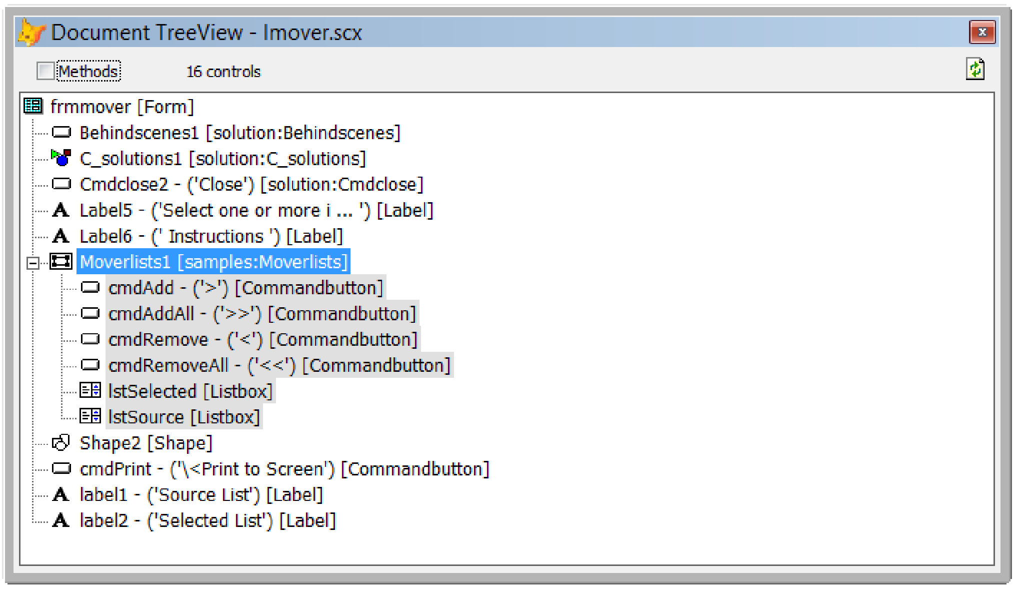
There are a number of ways to control what appears in Document Treeview. The two checkboxes above the treeview control offer three variations. When both are checked, you see all objects, as well as those methods that have code, as in Figure 36. If you uncheck All objects, you see only those objects that have any code, as well as the methods that contain code. Figure 38 shows Document Treeview for the lmover form with All objects unchecked. As you can see, this option makes it easy to see where there’s code. Finally, if you uncheck Methods, the All objects checkbox is hidden, and you see all the objects in the form or class. Figure 39 shows Document Treeview for the lmover form with Methods unchecked.


Document TreeView - lmover.scx

Methods ✓ All objects

```
frmmover [Form]
    Activate
    Deactivate
    Init
    cmdPrint - ('\<Print to Screen') [Commandbutton]
        Click
```

Figure 38. You can set Document Treeview to show only objects that contain code.

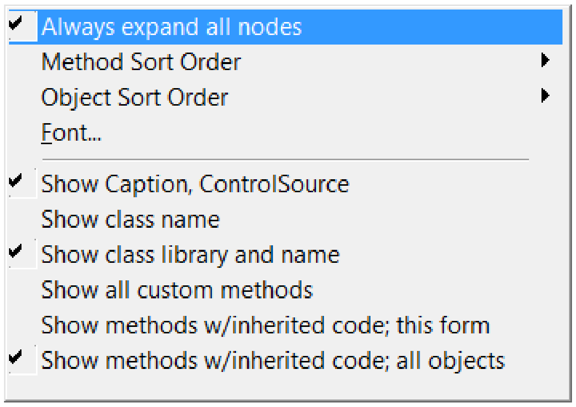

Document TreeView - lmover.scx

Methods 16 controls

```
frmmover [Form]
    Behindscenes1 [solution:Behindscenes]
    C_solutions1 [solution:C_solutions]
    Cmdclose2 - ('Close') [solution:Cmdclose]
    Label5 - ('Select one or more i ... ') [Label]
    Label6 - (' Instructions ') [Label]
    Moverlists1 [samples:Moverlists]
        cmdAdd - ('>') [Commandbutton]
        cmdAddAll - ('>>') [Commandbutton]
        cmdRemove - ('<') [Commandbutton]
        cmdRemoveAll - ('<<') [Commandbutton]
        lstSelected [Listbox]
        lstSource [Listbox]
    Shape2 [Shape]
    cmdPrint - ('\<Print to Screen') [Commandbutton]
    label1 - ('Source List') [Label]
    label2 - ('Selected List') [Label]
```

Figure 39. Unchecking the Methods checkbox tells Document Treeview to show all objects and no methods.

You can also control the display using the tool's context menu. Figure 40 shows the basic menu, as it appears when you right-click on the background of the tool. (Right-clicking on a node, of course, includes options specific to the node.) The first three items below the divider in Figure 40 determine what information is included for an object. The first item, Show Caption, ControlSource indicates whether you want to include the Caption or ControlSource of an object in the display, to help you determine which object it is. In Figure 39, you can see, for example, that the Caption for the button cmdClose2 is 'Close.' As you'd expect, the Show class name and Show class library and name options are mutually exclusive; checking one unchecks the other. However, you can choose to uncheck both and show no information about the class of an object.

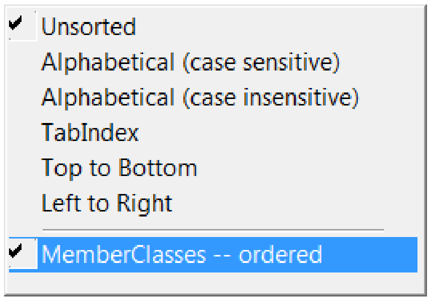


✓ Always expand all nodes
Method Sort Order ▸
Object Sort Order ▸
Font...
✓ Show Caption, ControlSource
Show class name
✓ Show class library and name
Show all custom methods
Show methods w/inherited code; this form
✓ Show methods w/inherited code; all objects

Figure 40. Document Treeview's context menu includes a number of ways to determine what's displayed.

The last three items in that section of the context menu determine which methods are displayed. By default, Document Treeview shows only methods that have code at this level of the hierarchy. Check Show all custom methods to also include all methods added to the current object. Check Show methods w/inherited code; this form to also show methods of the current object that have code higher in the inheritance hierarchy. Check Show methods w/inherited code; all objects to include methods of contained objects that have code somewhere in the inheritance hierarchy, as well.

The items above the divider control the appearance, but not the content of Document Treeview. The first item, Always expand all nodes, determines whether all items in the treeview are expanded when you open the tool or when you open a form or class with the tool open. The next two items determine the order in which methods and objects, respectively, appear in the list. For methods, the only choices are a case-sensitive alphabetical sort or a case-insensitive alphabetical sort. Objects offer a lot more choices, in addition to those two; the list is shown in Figure 41. You can leave them unsorted, in which case they appear in the same order as in the Property Sheet. You can also sort by TabIndex or from top to bottom or left to right.



✓ Unsorted
Alphabetical (case sensitive)
Alphabetical (case insensitive)
TabIndex
Top to Bottom
Left to Right
✓ MemberClasses -- ordered

Figure 41. There are a number of ways to sort objects in Document Treeview.

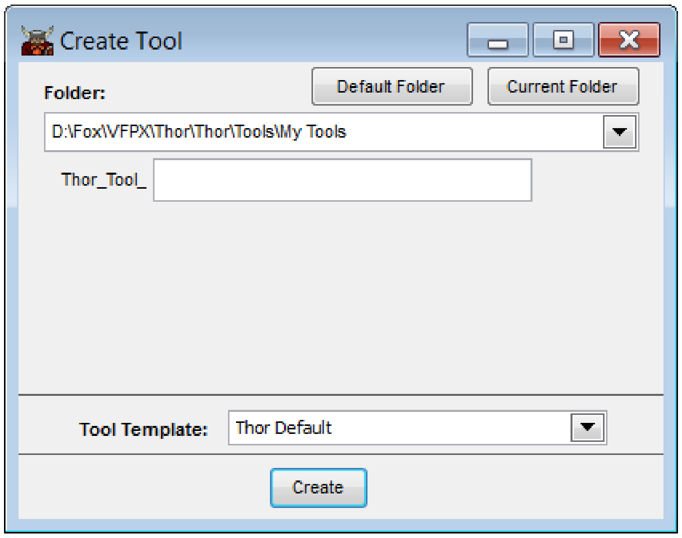
By default, objects within controls that have the MemberClass property sort in creation order, the same order in which they appear in the Property Sheet. Checking the final option in Figure 41, MemberClasses -- ordered, sorts them according to the order specified, such as PageOrder or ColumnOrder.

Adding tools to Thor

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 and fills a cursor with the names of all the form classes used in that project (that is, those on which at least one form is based). The whole thing is about 40 lines of code. The problem with these little tools is that when I want to use them, I have to find them and look at the code to remember how. 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 I describe above and stick them into the Thor Tools menu to keep them handy. Among other benefits, Thor manages the code so I 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 on the Tool Definitions tab. Click the Create Tool button to open the Create Tool dialog, shown in Figure 42. 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.



Folder: D:\Fox\VFPX\Thor\Thor\Tools\My Tools

Thor_Tool_

Tool Template: Thor Default

Figure 42. 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, slightly reformatted to fit the page. 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. Prompt is required and contains the prompt that will appear on the Thor Tools menu. Description appears only in the Thor configuration dialog.

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 = ''

    * 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 Source, Category and Sort properties let you specify where the tool appears in the Thor Tools menu. 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."

If Category is empty, the value in Source specifies the submenu on which the tool appears. You might use your initials or your company to group all of your own tools together.

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 the tool to get a list of form classes used.

Listing 3. The Thor properties set for the Get form classes tool.

```
    * Required
    .Prompt       = 'Get form classes' && used in menus

    * Optional
    Text to .Description NoShow && a description for the tool
Fill a cursor with names of the form classes used in a project
    EndText
    .StatusBarText = ''

    * These are used to group and sort tools when they are displayed in menus
    * or the Thor form
    .Source       = 'TEG' && 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
```

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 Get Form Classes tool. As you can see, it's not terribly complex. It checks for an active project, and if one is found, a cursor is created to hold the list of form classes. The code then loops through the files in the project. When a form file is encountered, it's opened as a table, and the form-level record found. If we've seen this form class before, it just counts this instance. If this is a new form class for our list, a record is added to the FormClasses cursor. After the loop is complete, the cursor opens in a BROWSE window.

Listing 4. The ToolCode procedure for the Get Form Classes tool.

```
  LOCAL cClassName, oFile, oProject, nOldSelect

  IF TYPE("_VFP.ActiveProject") = "U"
    MESSAGEBOX("No active project", 0+48, "Get form classes")
    RETURN
  ENDIF
```

```
oProject = _VFP.ActiveProject

nOldSelect = SELECT()

IF USED("FormClasses")
  USE IN SELECT("FormClasses")
ENDIF

CREATE CURSOR FormClasses (cClass C(30), nCount N(3))
INDEX on UPPER(cClass) TAG cClass

SELECT 0
FOR EACH oFile IN oProject.Files
  IF oFile.Type = "K"
    TRY
      USE (oFile.Name) ALIAS __Form
      LOCATE FOR UPPER(BaseClass) = "FORM"
      cClassName = __Form.Class
      IF SEEK(UPPER(m.cClassName), "FormClasses", "cClass")
        REPLACE nCount WITH nCount + 1 IN FormClasses
      ELSE
        INSERT INTO FormClasses VALUES (m.cClassName, 1)
      ENDIF
      USE IN DBF("__Form")
    CATCH
    ENDTRY

  ENDIF
ENDFOR

* Make sure last form was closed.
USE IN SELECT("__FORM")

SELECT FormClasses
BROWSE NOWAIT
```

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 menus and hotkeys. Once you do so, the new tool is included in the Thor Tools menu, as in Figure 43.

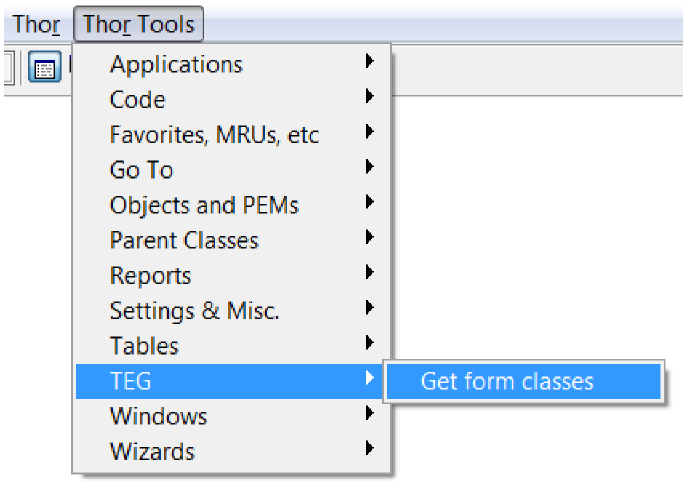
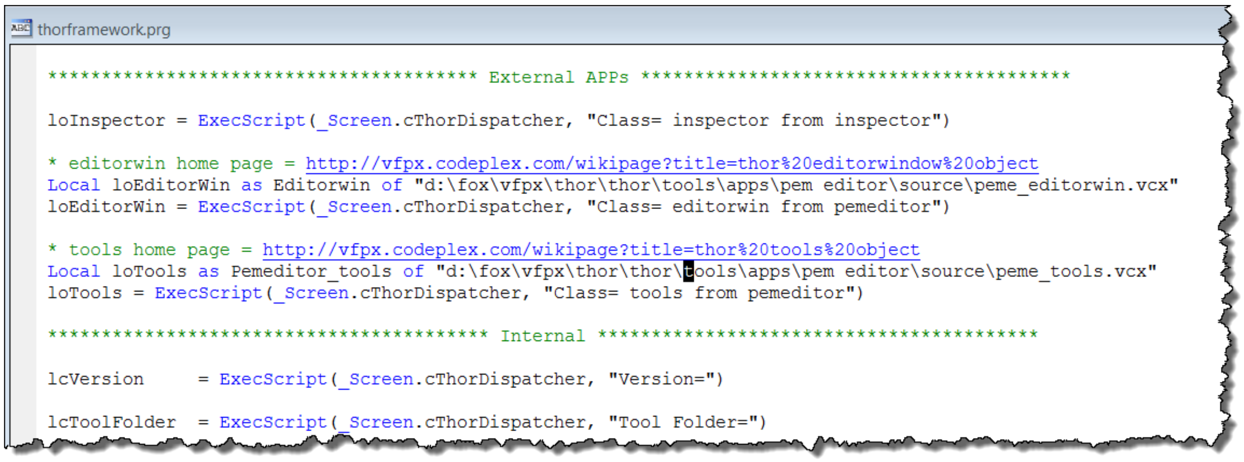


Figure 43. Once you finish the definition of a new tool and refresh Thor, the tool is shown on the menu.

Using Thor's capabilities in new tools

While you can write a new tool with standard VFP code (as in the Get Form Classes 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 | Thor Framework from the menu. A window opens containing code you can cut and paste into your tool code. Figure 44 shows part of the Thor Framework. (Be aware that the Thor Framework is smart enough to show the correct path for your installation. Figure 44 shows where the files are located on my computer.)



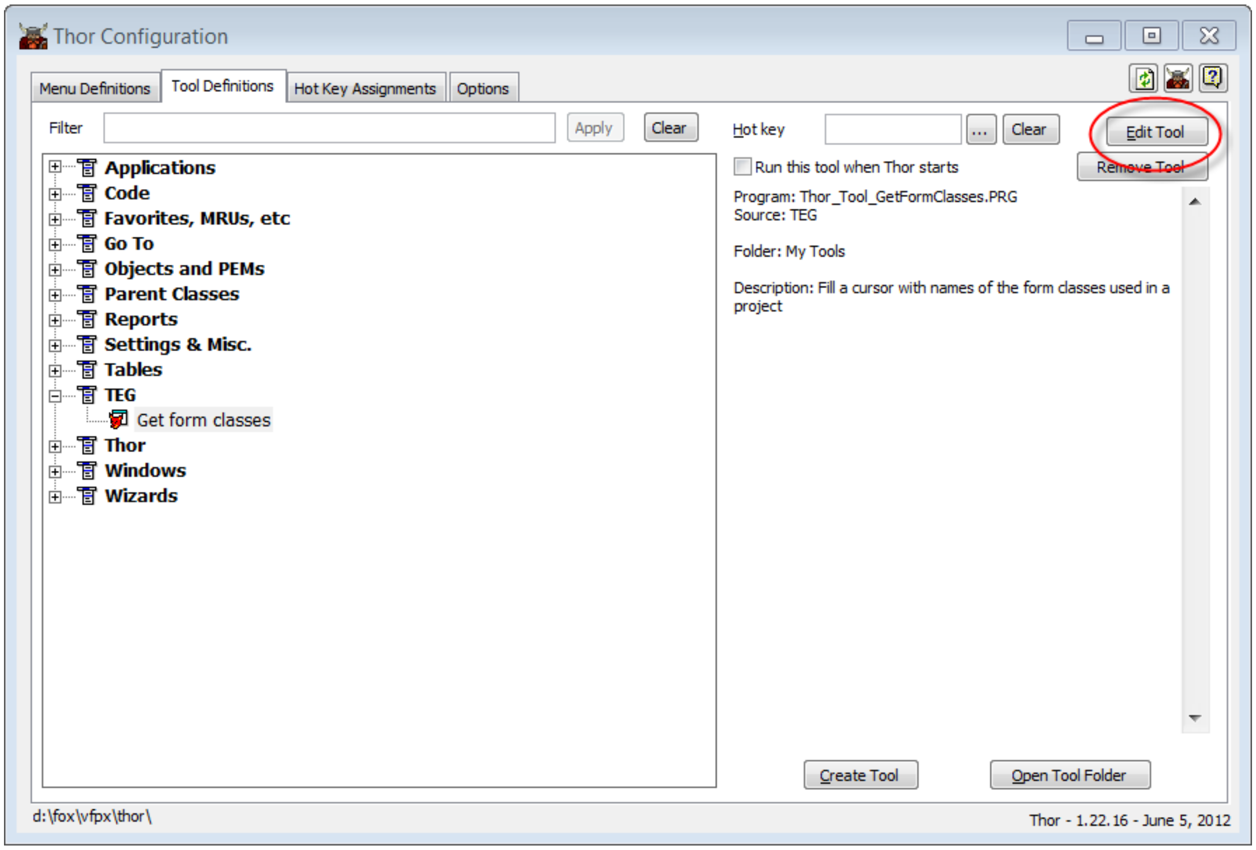
thorframework.prg

```
**************************************** External APPs ****************************************

loInspector = ExecScript(_Screen.cThorDispatcher, "Class= inspector from inspector")

* editorwin home page = http://vfpx.codeplex.com/wikipage?title=thor%20editorwindow%20object
Local loEditorWin as Editorwin of "d:\fox\vfpx\thor\thor\tools\apps\pem editor\source\peme_editorwin.vcx"
loEditorWin = ExecScript(_Screen.cThorDispatcher, "Class= editorwin from pemeditor")

* 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")

***************************************** Internal *****************************************

lcVersion     = ExecScript(_Screen.cThorDispatcher, "Version=")

lcToolFolder  = ExecScript(_Screen.cThorDispatcher, "Tool Folder=")
```

Figure 44. The Thor Framework lets you take advantage of code in Thor in your tools.

While a complete discussion of the Thor Framework is beyond the scope of this paper, I'll show a small example of how you can you use it. (See http://vfpx.codeplex.com/wikipage?title=Thor%20Tools%20Making%20Tools for more information about the Thor Framework.) The Get Form Classes tool described in the last section works only when you have a project open. One of the cool features of many Thor tools is that they can see what's under the mouse and operate on that item. That would be a handy capability for this tool—if there's no open project, then find the name under the mouse 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. 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, indicated in Figure 45. The form shown in Figure 46 appears. If all you want to do is see how the tool works, choose the second button, "View this file in Read-Only mode."



Run this tool when Thor starts

Program: Thor_Tool_GetFormClasses.PRG
Source: TEG

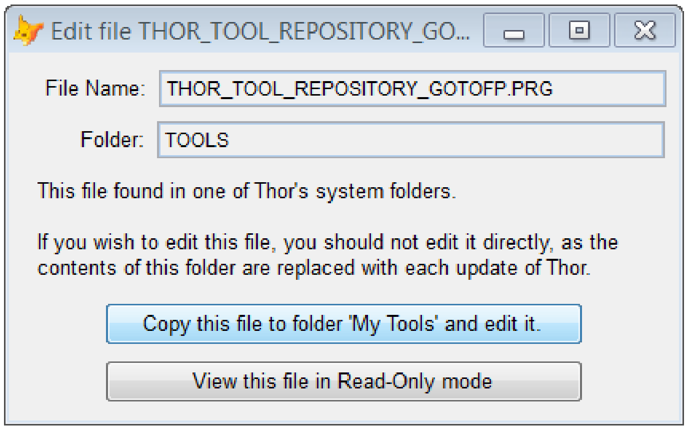
Folder: My Tools

Description: Fill a cursor with names of the form classes used in a project

d:\fox\vfpx\thor\

Thor - 1.22.16 - June 5, 2012

Figure 45. You can modify a tool by locating it in the Thor Configuration form and clicking Edit Tool.


Edit file THOR_TOOL_REPOSITORY_GO...

File Name: THOR_TOOL_REPOSITORY_GOTOFP.PRG

Folder: TOOLS

This file found in one of Thor's system folders.

If you wish to edit this file, you should not edit it directly, as the contents of this folder are replaced with each update of Thor.

Copy this file to folder 'My Tools' and edit it.
View this file in Read-Only mode

Figure 46. This form appears when you attempt to open any built-in Thor tool. To modify the tool, choose the first button. To simply look at its code, choose the second button.

I had to actually look at the PEM Editor source code to figure out how Thor can operate on the item named under the mouse. When I looked at the code for the SuperBrowse tool, I found the lines in Listing 5.

Listing 5. 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. Eventually, I found the line of code in Listing 6. 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 6. 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 Get Form Class tool. I opened the tool code as described above.

To set up the ability to read the text under the cursor, we need the Tools class from the Thor framework. Figure 44 includes the code to make that class available. You can just copy those three lines from the framework and paste them into the appropriate place in the ToolCode procedure. Once the Tools class is instantiated, we 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 7 (slightly reformatted to fit the page).

Listing 7. Replace the code to check whether a project is open with this code to allow the Get Form Classes 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()

    * Now find just the path and filename in the line.

    lProjWasOpen = .F.

    TRY
      MODIFY PROJECT (lcText) NOWAIT
      lSuccess = (TYPE("_VFP.ActiveProject") <> "U")
    CATCH
      lSuccess = .F.
    ENDTRY
  ELSE
    lSuccess = .T.
    lProjWasOpen = .T.
  ENDIF

  IF NOT m.lSuccess
    MESSAGEBOX("No active project", 0+48, "Get form classes")
    RETURN
  ENDIF
```

In testing, I found that this code works only when the specified project is in the path. I'm sure that the Thor Framework would let me read the whole project path under the cursor, but I haven't yet puzzled out exactly how.

Setting up options for a tool

As discussed in the "Create Locals" and "Add MDots to variable names" sections, earlier in this paper, some Thor tools let you customize them by setting options. The architecture for specifying options is open, so you can add options to existing tools or include them for your own tools.

The first step in setting up options is to add a class definition for each option to the program that implements the tool. These are only technically class definitions; their purpose is to provide a place to define each option. For each option, you need to specify four properties, described in Table 1. Listing 8 shows the class definitions that set up the options for the Add MDots tool.

Table 1. For each tool option, you need to give these properties values.

Property	Purpose
Tool	The name of the tool that the option affects.
Key	A string that uniquely identifies the option.
Value	The default value for this option.
EditClassName	The name of a class and class library that provides a user interface for specifying this option's value, in the form: <class> from <class library>

Listing 8. These three class definitions at the end of the code for the Add MDots tool specify three options (shown in Figure 29).

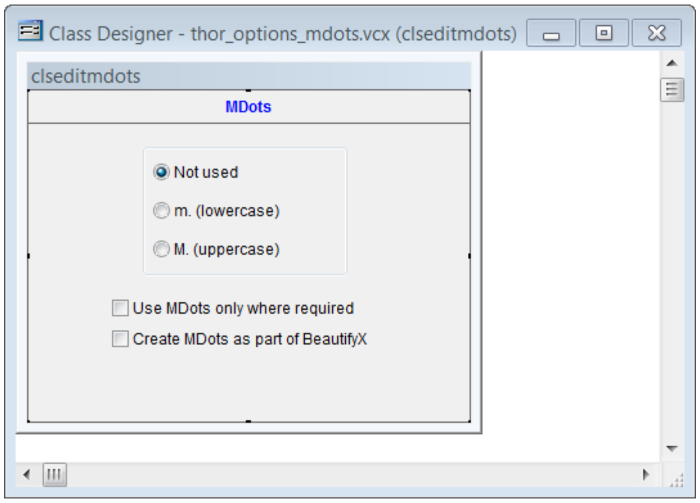
```
Define Class clsMDotsProperties As Custom

   Tool         = 'MDots'
   Key          = 'MDots Usage'
   Value        = 1
   EditClassName = 'clsEditMDots from Thor_Options_MDots.VCX'

Enddefine

Define Class clsMDotsWhereRequired As Custom

   Tool         = 'MDots'
   Key          = 'MDots where required'
   Value        = .F.
   EditClassName = 'clsEditMDots from Thor_Options_MDots.VCX'

Enddefine

Define Class clsMDotsinBeautifyX As Custom

   Tool         = 'MDots'
   Key          = 'MDots in BeautifyX'
   Value        = .F.
   EditClassName = 'clsEditMDots from Thor_Options_MDots.VCX'

Enddefine
```

The next step in providing options is to list the classes in the definition part of the program that implements the tool. Fill the OptionClasses property with a comma-separated list of the classes, as in Listing 9.

Listing 9. This line in the top (definition) portion of the Add MDots tool indicates that the tool has three options, defined by the classes listed.

```
.OptionClasses = 'clsMDotsProperties, clsMDotsWhereRequired, clsMDotsinBeautifyX'
```

Next, you need to create the UI class to display the options. This is the class specified in the EditClassName of each option. As the example indicates, you can (and should) use a single class to hold the UI for multiple options for a single tool. The class should be based on the Container base class. Figure 47 shows the clsEditMDots clas used for the Add MDots tool.



● Not used
○ m. (lowercase)
○ M. (uppercase)

☐ Use MDots only where required
☐ Create MDots as part of BeautifyX

Figure 47. This class, based on the Container class, contains controls for setting options for the Add MDots tool.

At runtime, the specified class is added to another container class (and resized appropriately). That container class has two methods to let you 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.

Call these methods as needed to set up the Options page and to save the user's choices. For example, the Refresh method of the first checkbox in Figure 47 contains the code in Listing 10, while the InteractiveChange method saves the user's choice, with the code in Listing 11.

Listing 10. This line, in the Refresh method of the checkbox, ensures that the control reflects the current setting for the option.

```
This.Value = This.Parent.Parent.GetOption('MDots where required', 'MDots')
```

Listing 11. This code, in the checkbox's InteractiveChange method saves the user's choice.

```
This.Parent.Parent.SetOption('MDots where required', 'MDots', This.Value)
```

The last step in providing options is to use the options in the tool's code. You can do that in the ToolCode procedure or in code that procedure calls. You need to instantiate the Thor engine and call its GetOption method (with the same parameters as shown above). The code in Listing 12 retrieves the value of the "MDots where required" option and saves it to a variable, so it can be used.

Listing 12. To apply the user's selected option, you instantiate the Thor engine, and call the GetOption method.

```
loThor      = Execscript (_Screen.cThorDispatcher, 'Thor Engine=')
llMDotsWhereRequired  = loThor.GetOption ('MDots where required', 'MDots')
```

Give Thor a try

Changing your work habits is hard, but when a new tool offers real benefits, it's worth the effort. In my view, Thor offers more than enough benefits to make it worth adding to your development environment. Once you figure out which tools you're likely to use often, you can assign them hot keys or put them in pop-up menus to provide easy access.

Not only does Thor provide you with a few dozen handy tools, but it lets you grab all those little tools you've written over the years and make them easily available.