Top 10 (or more) Reasons to Use the Toolbox

Tamar E. Granor
Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Voice:215-635-1958
Email: tamar@tomorrowssolutionsllc.com

The VFP Toolbox is a terrific tool, but many VFP developers have never really used it. In this session, we'll look at why you should use this tool everyday. We will look at built-in capabilities like an easy way to drop controls into grid columns. We'll also see how to customize it for your development environment, including having each control you drop given an appropriate name. Finally, we'll explore some easy extensions that make the Toolbox even more useful, such as making it easy to store web links.

VFP 8 included three new developer tools: Code References, the Task Pane Manager, and the Toolbox. While many developers have integrated Code References into their everyday toolkits, few use the Task Pane Manager or the Toolbox regularly. For the Task Pane Manager, that may be a reasonable choice, as it hasn't lived up to its initial promise. (Specifically, few additional task panes have come out of the VFP community.)

However, the Toolbox is a useful, powerful tool that simplifies many of the tasks of building forms and classes, and does much more besides. It's easier to use and more extensible than the Form Controls toolbar that it replaces.

A brief history of putting controls onto forms

When the Form Designer replaced FoxPro 2.x's Screen Builder in VFP 3, it included three ways to put controls on forms and classes.

The most basic and visible way to drop controls is the Form Controls toolbar. By default, it opens automatically when you open the Form Designer or Class Designer; to drop a control, you click on the control in the toolbar, and then click where you want to put it. This toolbar was so "in your face" that many people never looked any further.

The Project Manager offers a second approach. Switch to the Classes tab or expand the Class Libraries section in the All tab, then expand the relevant library and you can drag a control and drop it where you want it. Many VFP developers settled on this as their standard way of creating forms and classes.

There was a third choice, as well. The Class Browser is a separate tool actually written (by Ken Levy) in VFP code. It lets you open one or more class libraries (or even an entire project) and explore their contents. You can drop a control onto a form or class by clicking the class in the Class Browser, then dragging the icon in the upper left corner to the position where you want it. (In fact, that approach even works on live forms, so you can do the cartoon trick of creating a blank form, dropping live controls onto it, and then using those controls.)

While all three techniques work, each of them has quirks and drawbacks, things that make it clumsy or inefficient or cases it doesn't handle. I won't catalog all of the issues with the old tools here, but I will list the main ones that bother me.

With the Form Controls toolbar, the biggest issues relate to how you get your own class libraries shown. The toolbar can show either the native VFP controls, all registered ActiveX controls, or the controls in a single class library. Although the toolbar does remember how you left it, when you close VFP and reopen it, any class libraries other than the one currently displayed are removed from the list.

You can register class libraries on the Controls page of the Tools | Options dialog to make them available in every VFP session, but there's no easy mechanism for setting up different lists of libraries for different projects.

Another problem is that, by default, all controls based on the same base class look the same, and you have to rely on their tooltips to figure out which is the one you want. In my experience, the order changes when you edit classes in the library, so you can't even rely on their position. (You can change the icon for a class using the Class Info dialog that's available from the Class Designer, but in my experience, very few people ever do so.)

All three tools share another big weakness; there's no easy way to create new forms based on a custom form class. For me, this is a fairly common activity when developing and the workarounds for it (registering a form class in the Tools | Options dialog, or using the CREATE FORM command with the AS clause) are clumsy; registering a form class in the Tools| Option dialog doesn't address the idea that you may need to work with several form classes.

VFP 5 brought the Component Gallery, an alternate face for the Class Browser that offers a way to organize classes and much more into catalogs based on your own organizational structure rather than where they live or what project they belong to. It's an incredibly flexible tool, with tremendous extensibility built in. However, it's also hard to understand and use and never gained much traction in the community.

The bottom line for me, at least, is that I never found a really comfortable way to add controls to forms and classes. So when VFP 8 brought the Toolbox, I was happy to give it a try and see what it offered. I quickly settled on the Toolbox as my standard way of handling this task; it's actually capable of much, much more.

Toolbox overview

Before looking at my list of reasons for using the Toolbox, let's start with a quick overview of basic Toolbox functionality.

The Toolbox (shown in Figure 1) contains *categories*, represented by the raised horizontal bars. Each category contains one or more *items*, the sunken bars. An item can be a class, an ActiveX control, a web service, a text scrap, a script or a file.

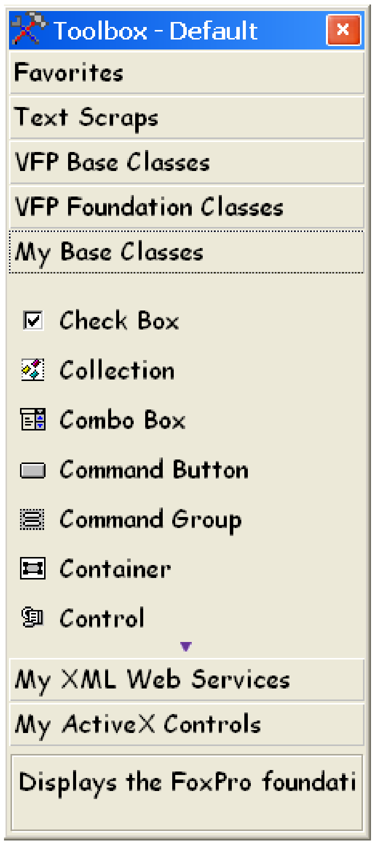
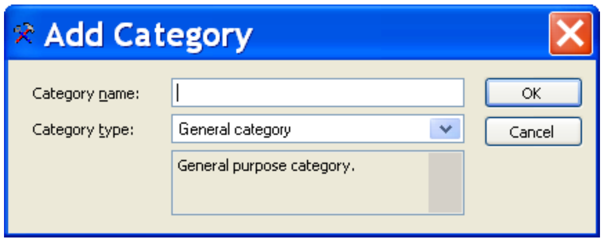


Figure 1. The Toolbox contains categories (the horizontal "buttons") and items. This figure shows the built-in categories. You can add your own, as well.

Items can be dropped onto forms and classes, or into code windows. The exact result depends on the type of item and where you drop it. The most common use is dropping classes onto forms or other classes. You can either drag-and-drop or double-click.

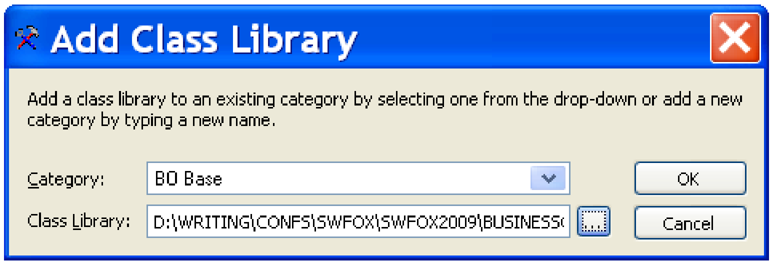
Adding your own categories and items is easy. To add a category, right-click and choose Add Category. In the Add Category dialog (Figure 2), specify a name and the type of category. To set up a category to contain VFP classes, use the default General category.



Category name:
Category type: General category
General purpose category.

Figure 2. Use this dialog to add a new category. For VFP classes, choose the General category type.

To add classes to the Toolbox, open the category you want to add them to (by clicking on it). Then right-click inside the category and choose Add Class Library. The Add Class Library dialog (Figure 3) appears. Point to the class library and click OK.


Add Class Library

Add a class library to an existing category by selecting one from the drop-down or add a new category by typing a new name.

Category: BO Base

Class Library: D:\WRITING\CONFS\SWFOX\SWFOX2009\BUSINESS(

Figure 3. To add classes to the Toolbox, choose a category and point to the class library.

Figure 4 shows the Toolbox with the newly added category and classes. Once you add a class library to the Toolbox, you can add its classes to forms and other classes just like the classes that are built into the Toolbox.

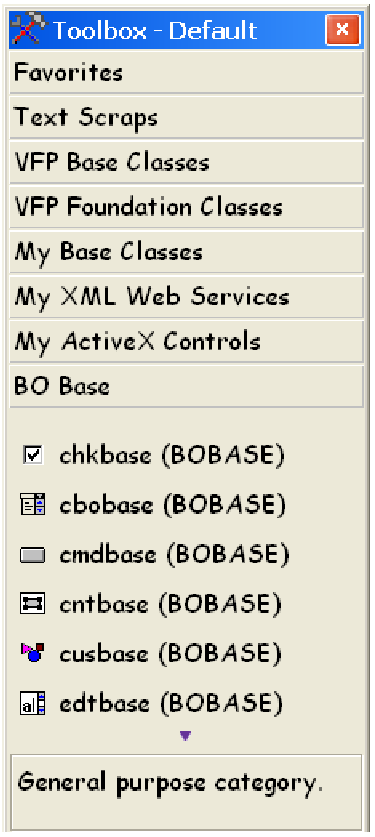


Figure 4. Classes you add to the Toolbox can be used just like the built in classes.

This is just a tiny piece of what the Toolbox can do. To learn more about its basic capabilities, see the papers referenced in "Make the Toolbox work for you" at the end of this document.

Reasons that use the Toolbox as is

As it ships, the Toolbox is useful and powerful. The first group of reasons to use it work with the Toolbox as is, though some involving customizing it through its own interface. Later in this document, I'll look at reasons to use the Toolbox that involve extending it.

Reason 1a: Drop controls into containers

One of the tasks that's annoying with all the other techniques is adding controls inside a container, such as a pageframe or the Container class itself. To do so, you have to right-click on the container, and choose Edit, then drop the desired control. If the container you're interested in is several levels down in the hierarchy, you have to repeat this sequence for each level. (In fact, in recent versions of VFP, you can jump right to the innermost container by using Ctrl+Shift+Click.)

The Toolbox makes this much easier. Just drag and drop a control onto the container you want. There's no need to select the container first; the Toolbox is smart enough to figure out what container you dropped on and add the new control to that container. It's even smart enough to know that if you drop a control on top of another non-container control within a container, the new control should be added to the container.

Figure 5 demonstrates dropping a control into a container. Note that nothing on the form is selected. Figure 6 shows the result; now the new control is selected and the various containers show the usual "edit me" outlines.

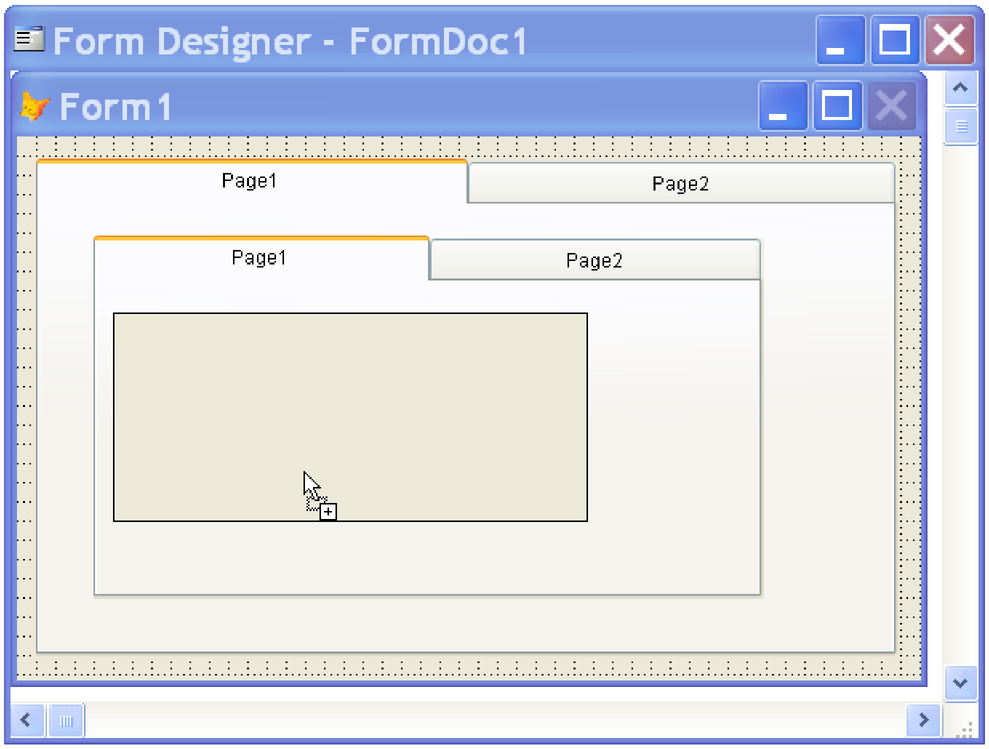


Figure 5. The Toolbox lets you drop controls onto containers without drilling down to the container first.

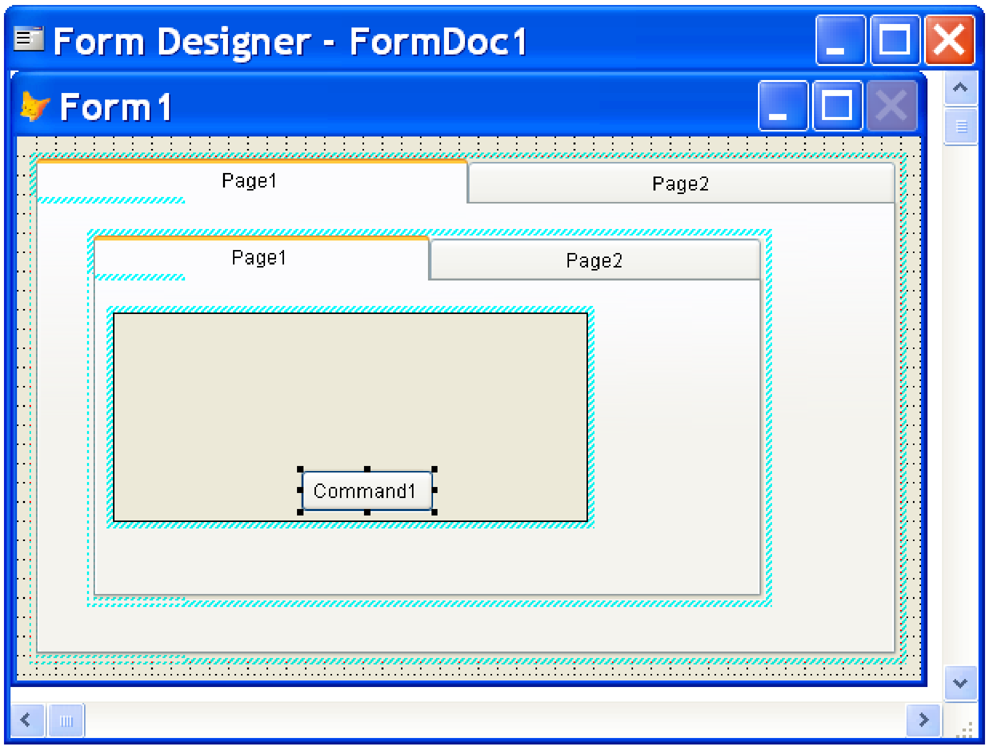


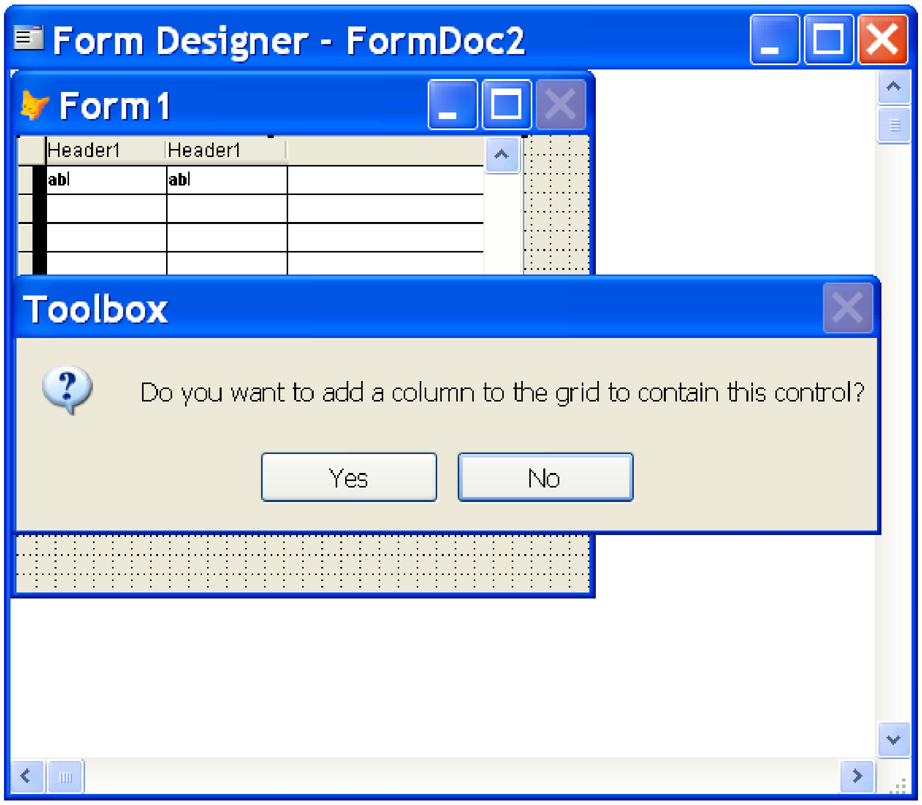
Figure 6. After you drop a control into a container, the control is selected.

Reason 1b: Drop controls into grids

Grids are a special case of a container control, and getting the right controls into grid columns has always been tedious using the earlier tools. As with other containers, you have to select the right column for editing, and then drop the control on it. But what has made this task particularly aggravating is that columns can hold multiple controls, so adding a control to a column doesn't remove the textbox that's there by default. Removing that textbox requires choosing it in the Property Sheet, then clicking on the title bar of the form or the Form Designer, then pressing the Delete key.

The Toolbox makes adding controls to grid columns and removing the default textbox a breeze. Once you add a grid to a form or class, select the grid, and you can start adding controls by double-clicking on them or dragging and dropping them. There are three distinct behaviors, depending on the situation.

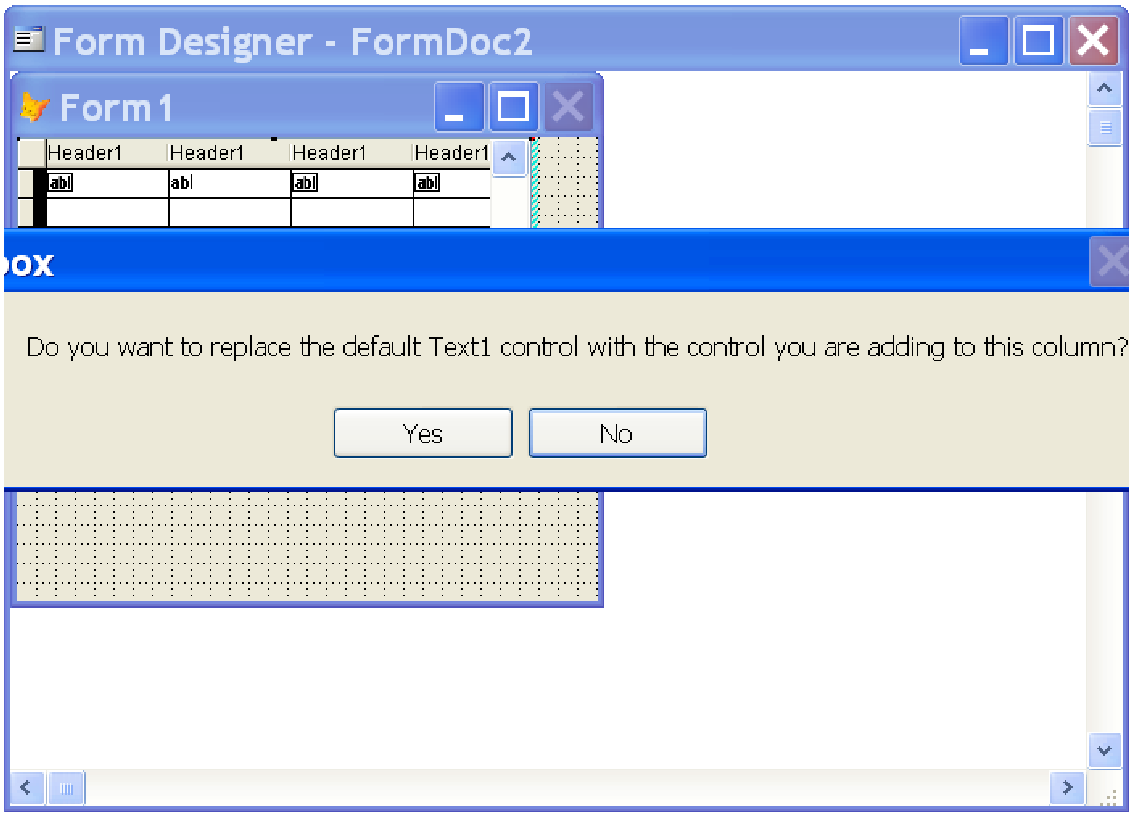
If you double-click on a control in the Toolbox with a grid selected, you're prompted to add a column to the grid to hold that control, as in Figure 7. If you drag and drop a control to an empty space in the grid, you see the same behavior.



Do you want to add a column to the grid to contain this control?

Figure 7. You can add a new column to a grid by double-clicking on the appropriate control in the Toolbox.

But if you drag and drop a control into an existing column, the behavior you see depends on what's already there. If the column contains a textbox named Text1, you can replace that control with the one you're dropping, as in Figure 8. (You control whether that message appears, through a Toolbox option.)



Do you want to replace the default Text1 control with the control you are adding to this column?

Figure 8. When you drop a control into a column containing only a textbox named Text1, you can remove the textbox at the same time.

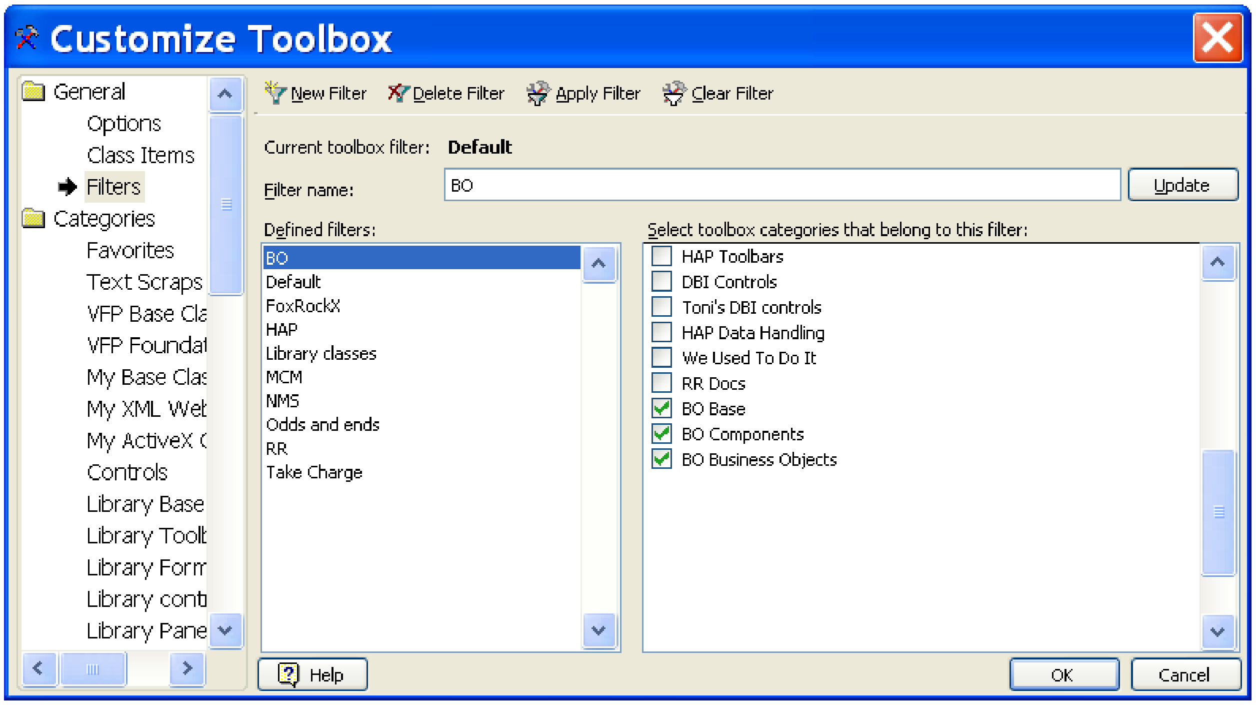
If the column is empty or contains any other controls, the control you drop is added to the column. You can determine through one of the Toolbox's options whether the newly added control automatically becomes the CurrentControl for the column.

Reason 2: Easy access to classes for a project

One of my issues with the Form Controls toolbar is that all registered class libraries show up, making it hard to be sure you're using the right classes for a given project. The Toolbox lets you filter what's shown, so you can look at only the classes you should for a given project. A filter contains a subset of the categories defined.

To create a filter, right-click on the Toolbox and choose Customize Toolbox. Then click on Filters in the General section of the left-hand list. Then, click New Filter on the button bar at the top. In the Filter name textbox, specify the name for your filter and click Update to update it in the list below.

Then, check each category you want to include in this filter. Figure 9 shows the dialog after creating and naming a new filter and adding some categories to it.


Customize Toolbox

General
Options
Class Items
Filters
Categories
Favorites
Text Scraps
VFP Base Cla
VFP Foundat
My Base Clas
My XML Wet
My ActiveX (
Controls
Library Base
Library Toolt
Library Form
Library conti
Library Pane

New Filter | Delete Filter | Apply Filter | Clear Filter

Current toolbox filter: **Default**

Filter name: BO

Defined filters:
- BO
- Default
- FoxRockX
- HAP
- Library classes
- MCM
- NMS
- Odds and ends
- RR
- Take Charge

Select toolbox categories that belong to this filter:
- [ ] HAP Toolbars
- [ ] DBI Controls
- [ ] Toni's DBI controls
- [ ] HAP Data Handling
- [ ] We Used To Do It
- [ ] RR Docs
- [✓] BO Base
- [✓] BO Components
- [✓] BO Business Objects

Figure 9. Filters let you show only a subset of the defined categories.

Any category can appear in as many filters as you want, and you can define many filters. I use one filter for the default Toolbox contents (important to me because I do demonstrations of the Toolbox), and one for each project I'm working on. (I also name my categories using an abbreviation for the relevant project, so it's easy to see which categories belong to which project.)

Once you've created filters, a Filter item appears in the Toolbox's context menu; it has a sub-menu listing all the defined filters. Choose one and the Toolbox shows only the categories included in that filter. Figure 10 shows the Toolbox after selecting the newly-defined BO filter.

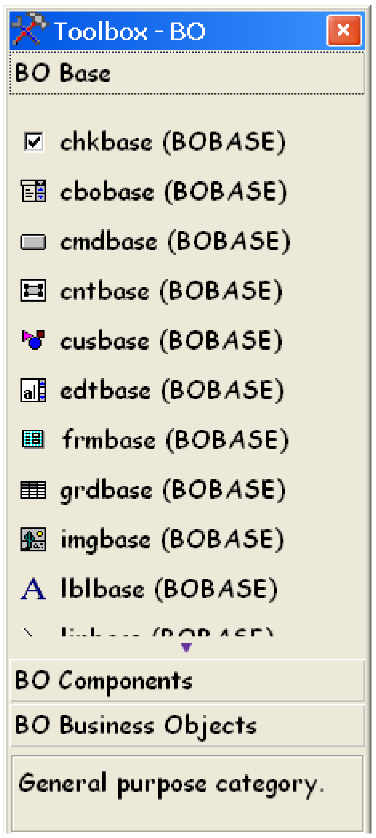


Figure 10. When you apply a filter, only the categories in that filter appear. This makes it easier to see what you have, and helps to use the right classes for each project.

Reason 3: Create forms from form classes

Whether you're working with the Form Controls toolbar, the Project Manager, or the Class Browser, creating a new form from a form class isn't handled by the tool. If all the forms are to be created from a single class, that's not a big problem. You can specify a form "template class" on the Forms page of the Options dialog. Then, when you issue CREATE FORM, the new form will be based on that class.

But in my projects, I tend to have a few different form classes. Typically, there's a "base" form class, then subclasses of that one for dialogs, for data entry forms, and for reporting. (Sometimes, of course, additional subclasses are called for, as well.) So a single form template class doesn't do me any good.

With the Toolbox, it's easy. Right-click on the form class and choose Create Form, as in Figure 11. The Form Designer opens with the new form based on the specified class.

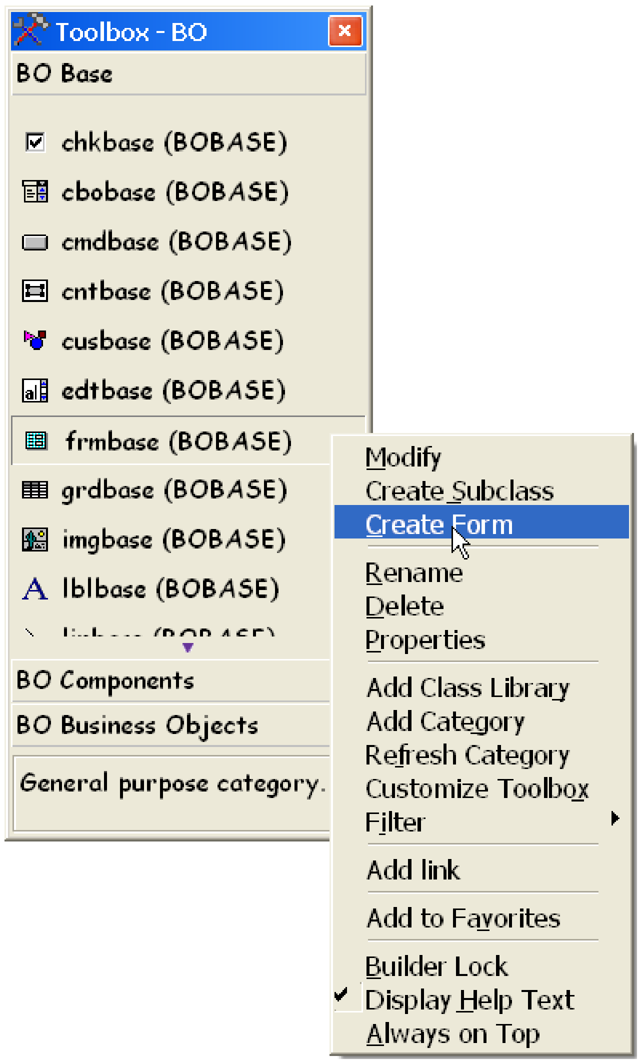


Figure 11. Creating a form based on a class is a simple operation in the Toolbox.

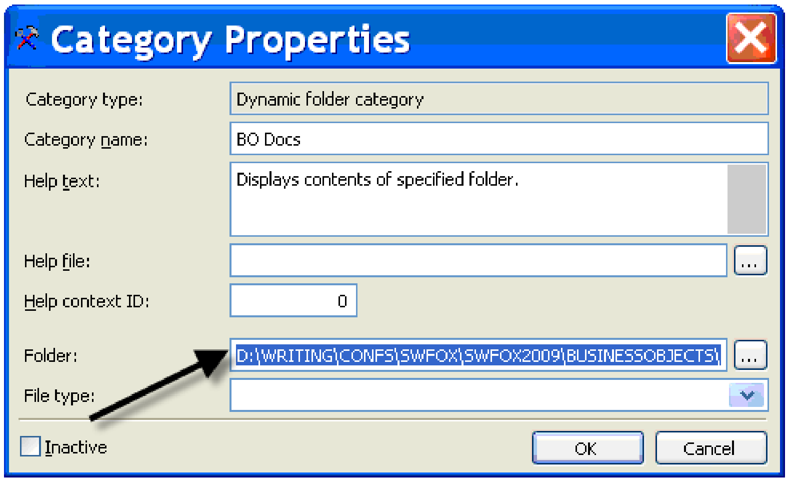
Since I tend to keep all the form classes for a project (other than the "base" form class) in a single class library, this is really easy for me.

Reason 4: Easy access to project documents

So far, we've only used the Toolbox for classes and forms. But in fact, it can hold all kinds of files, and provides one-click access to file content. One of the ways to take advantage of this is to create a category for the documents related to a project, such as specs, user guides, and so forth.

When a Toolbox item represents a file, clicking on that item opens the file in its native application. So you can put documents, spreadsheets, diagrams, and so forth into the Toolbox, and open them as needed.

There are two ways to set such a category up. If all or most of the documents you want to include are in a single directory, create the category as a Dynamic folder category. When you create such a category, the Category Properties dialog (Figure 12) opens and you specify the folder to look in. You can also indicate which file types to include by choosing from the dropdown or by typing a list of file skeletons. Later in this paper (Reason 10), I'll show you how to add your sets of skeletons to the dropdown.


Category Properties

Category type:	Dynamic folder category
Category name:	BO Docs
Help text:	Displays contents of specified folder.
Help file:	
Help context ID:	0
Folder:	D:\WRITING\CONFS\SWFOX\SWFOX2009\BUSINESSOBJECTS\
File type:	

Figure 12. For a Dynamic folder category, you specify the folder to look at and which types of files to include.

Once you click OK, the category is added and includes all files in that folder with the specified file types. However, you can still add other files to the category manually.

The alternative to a Dynamic folder category for this type of data is to use a General category and then add all the relevant files manually. To add a file to a category (whatever type of category it is), use the Customize Toolbox dialog. Click on the category to which you want to add the file, and then click Add Item. The Add Item dialog opens (as in Figure 13). To add a file so that it opens in its native application, choose the File type, then point to the file in the Open dialog that appears.

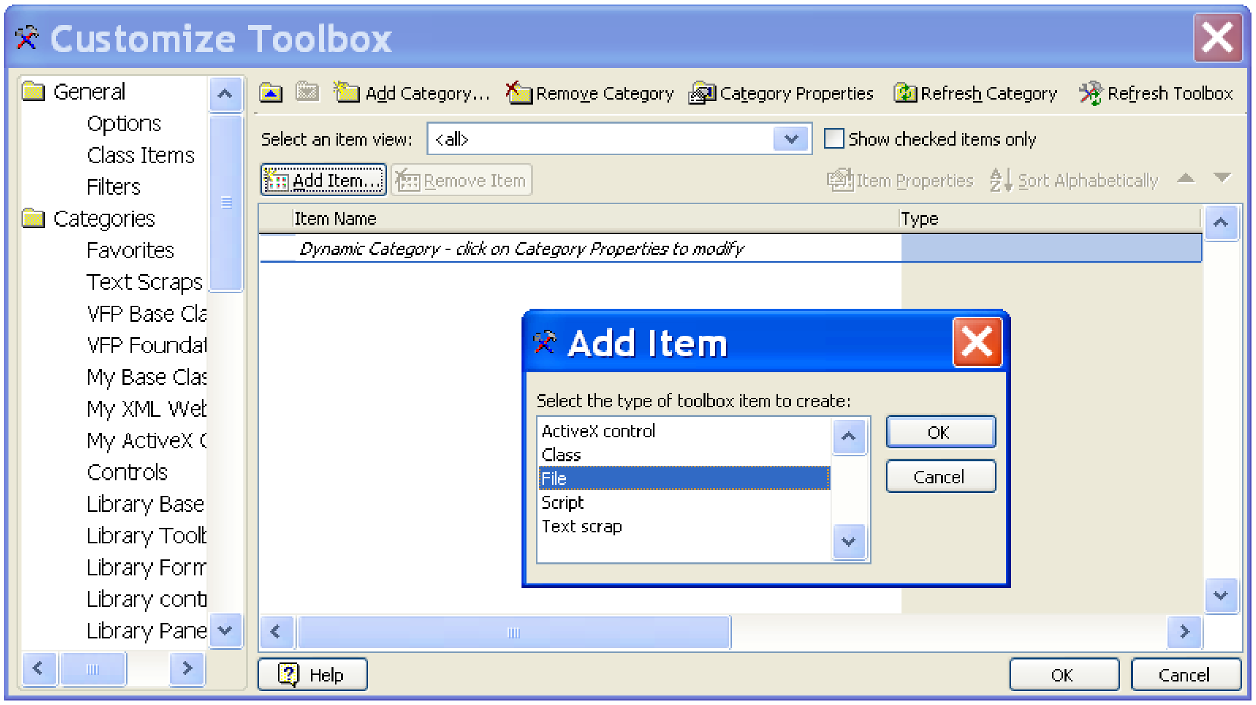


Figure 13. To add files to any type of category, click Add Item in the Customize Toolbox dialog.

Figure 14 shows two Word documents in a category. Note that File items are shown as links, since clicking on them opens the file in its native editor. You can change this behavior using the Customize Toolbox dialog, One of the items in the Options sections lets you require a double-click to open a file item.

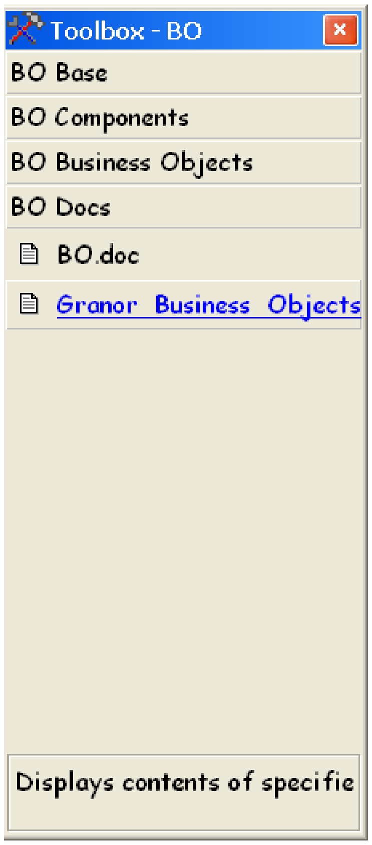
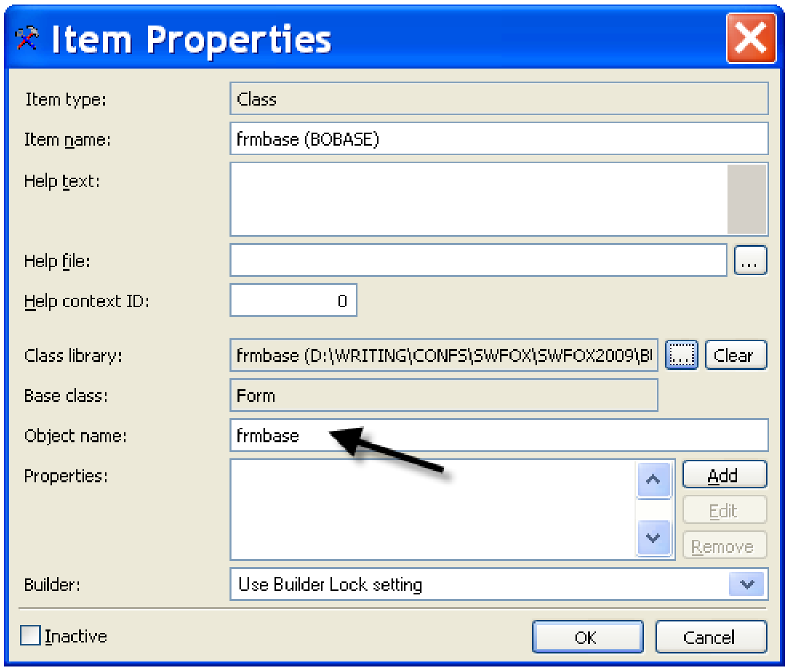


Figure 14. File items are links, so clicking opens the file in its native editor.

Reason 5: Make sure every control gets a name (and more)

Giving every control on a form or class a name that reflects its purpose is a best practice. But it's an easy step to miss when you're busy laying out a form or class. Even if you don't forget, having to switch over to the Property Sheet and do it for each control is tedious.

Toolbox to the rescue. In fact, the Toolbox offers two capabilities on this front. First, you can specify a "base" name for each class in the Toolbox, that is, the name from which the object name is created. You do so in the Item Properties dialog, as shown in Figure 15. When a control is dropped onto a form or class, its name is set to the specified object name followed by a number. (When you drop a control into a code window, the code uses the actual object name specified for that class without suffixing it.)


Item Properties

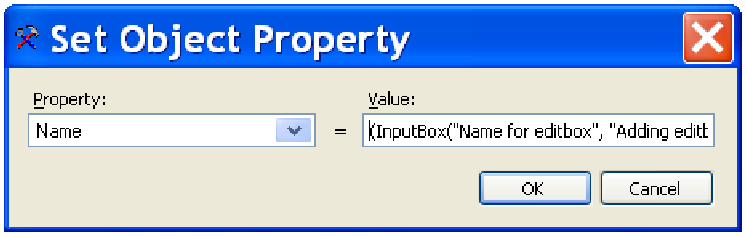
Item type:	Class
Item name:	frmbase (BOBASE)
Help text:	
Help file:	
Help context ID:	0
Class library:	frmbase (D:\WRITING\CONFS\SWFOX\SWFOX2009\BI
Base class:	Form
Object name:	frmbase
Properties:	
Builder:	Use Builder Lock setting

Figure 15. You can specify the "base" name for new controls from a class in the Item properties dialog.

While this makes a good first step, you can actually get prompted for the name as part of adding the control. The Toolbox has the ability to set instance properties, that is, properties of the control you're adding, much the way a Builder does.

To set it up, click the Add button in the Item Properties dialog. The Set Object Property dialog (Figure 16) appears. In the dropdown, choose the property you want to set, in this case, Name. In the textbox, specify the value to set that property to. What makes this ability so useful is that the "value" doesn't have to be a constant. In this case, it's an expression that calls the InputBox() function, so you can enter the name you want. Note that the parentheses around the expression are required in order to have the expression evaluated before assigning it, like this:

```
(InputBox("Name for editbox", "Adding editbox", ""))
```


Set Object Property

Property: Name = Value: (InputBox("Name for editbox", "Adding editt

Figure 16. You can set up the Toolbox to prompt you for the name for a control when you add it to a form or class.

When you drop this editbox class onto a form or class, either by drag or drop or by double-clicking, you see the dialog in Figure 17. Type the name you want and the control is added with that name. No need to go to the Property Sheet to specify the control's name.

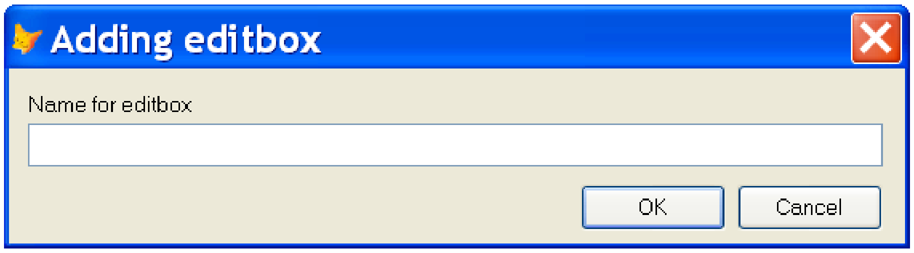


Figure 17. After setting up the prompt for the object's name, this dialog appears each time you drop this class onto a form or class.

You can use the same trick to set a control's Caption as you add it and, in fact, you can set many properties this way. However, don't use this ability as a substitute for subclassing. If you need a control that always has certain properties set to certain values, create a class with those properties set.

The tedious part about this reason is adding the InputBox() call to each class when you add a class library to the Toolbox. It wouldn't be terribly hard to automate that process, either by modifying the code that adds a class library to the Toolbox, or by adding an item to the Toolbox menu (see Reason 10).

Reason 6: "IntelliSense" for mousers

Starting in VFP 7, IntelliSense has provided an easy way to set up all kinds of typing shortcuts. Many are built-in, like MC for MODIFY COMMAND, and adding your own is easy, too. I have quite a few in my IntelliSense table, including MP for MODIFY PROJECT and BL for BROWSE LAST, both commands I use a lot.

IntelliSense also offers a way to run a chunk of code and insert the result. For example, the built-in ZLOC script pops up a list of defined local variables; when you choose one, that variable is inserted into the code window at the point where you typed ZLOC. You can add your own scripts as well. For example, I use a script called TEGMOD to insert a change comment that includes my initials and the date. Listing 1 shows the code (leveraged from a similar script created by Doug Hennig).

Listing 1. This IntelliSense script inserts a change comment when I type TEGMOD.

```
lparameters toFoxCode
local lcReturn

if toFoxCode.Location <> 0
   toFoxCode.ValueType = 'V'
   lcReturn = GetText()
```

```
endif toFoxCode.Location <> 0
return lcReturn

function GetText
local lcText, nDay, cMonthName, nYear, dToday
dToday = date()
nDay = day(dToday)
cMonthName = cmonth(dToday)
nYear = year(dToday)
text to lcText textmerge noshow
* Modified <<nDay>>-<<cMonthName>>-<<nYear>> by TEG
* ~
endtext
return lcText
```

As you can see, writing IntelliSense scripts is a bit trickier than adding shortcuts.

More importantly, for those more comfortable with the mouse than the keyboard, IntelliSense scripts aren't terribly appealing. The Toolbox offers the equivalent of IntelliSense scripts, but triggered by the mouse rather than a keystroke sequence. In fact, two kinds of Toolbox items offer alternatives to IntelliSense scripts.

The simpler form is a Text Scrap. The Toolbox comes with two built-in text scraps (the indicated items in Figure 18). One is a template for reporting bugs in the format the VFP team prefers. The other provides a comment block for the top of a program file; an example is shown in Figure 19.

A text scrap consists of a block of text, optionally containing textmerge expressions. For example, the definition of the text scrap for the Comment Header template is shown in Listing 2.

Listing 2. This text scrap, which comes with the Toolbox, creates a comment block appropriate for the top of a PRG file.

```
*!* Program:
*!* Author:
*!* Date: <<datetime()>>
*!* Copyright:
*!* Description:
*!* Revision Information:
```

In this example, everything is fixed except that textmerge is used to fill in the date and time when the scrap is dropped.

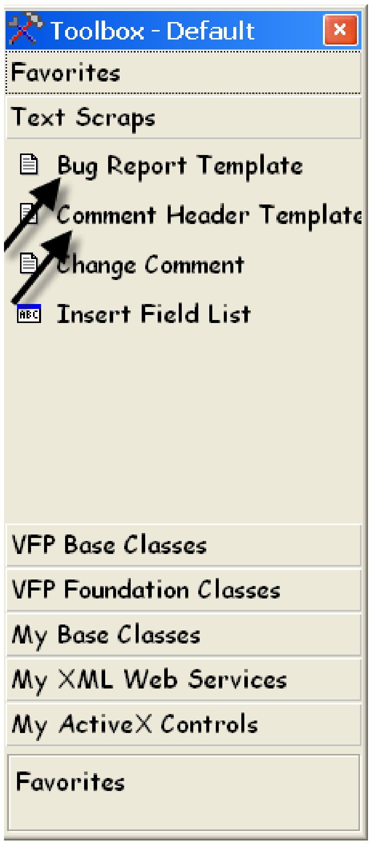
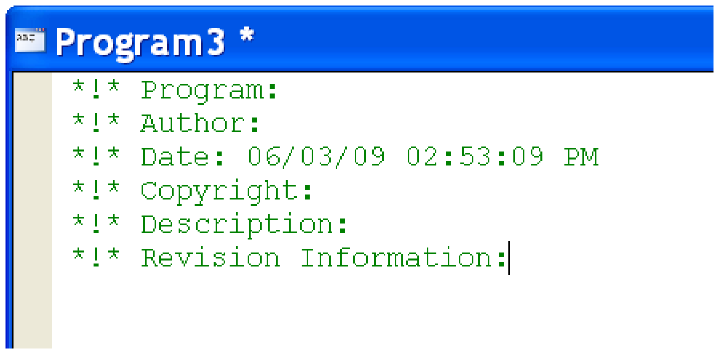


Figure 18. The Toolbox comes with two pre-defined text scraps.



```
*!* Program:
*!* Author:
*!* Date: 06/03/09 02:53:09 PM
*!* Copyright:
*!* Description:
*!* Revision Information:
```

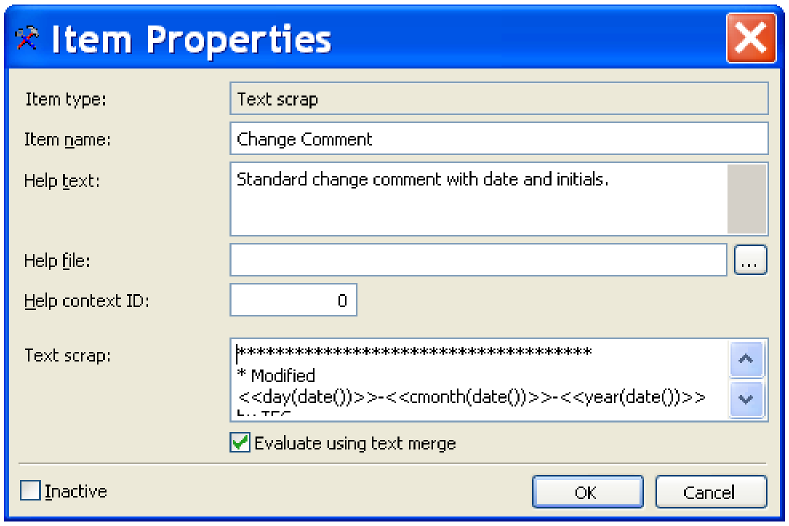
Figure 19. The Comment Header Template text scrap sets up the kind of block comment you'd put at the top of a program file.

To add a text scrap with this kind of behavior, click Add Item in the Customize Toolbox dialog. Then, choose Text Scrap from the Add Item dialog. The Item Properties dialog appears, set up for a text scrap. Specify the name for your text scrap, and in the Text scrap editbox, put the text or code to be inserted. If the scrap includes expressions to be evaluated when you drop it, make sure to check the Evaluate using text merge checkbox. Figure 20 shows the Item Properties dialog for my Change Comment text scrap. The result is the same as for my TEGMOD IntelliSense script (shown in Listing 1), but the code (in

Listing 3) is much simpler. To insert this comment, just drag it and drop it where you want the comment.

Listing 3. Setting up a change comment as a text scrap is much easier than writing an IntelliSense script.

```
**************************************
* Modified <<day(date())>>-<<cmonth(date())>>-<<year(date())>> by TEG
*
```



Item type: Text scrap
Item name: Change Comment
Help text: Standard change comment with date and initials.
Help file:
Help context ID: 0
Text scrap:
```
**************************************
* Modified
<<day(date())>>-<<cmonth(date())>>-<<year(date())>>
```
☑ Evaluate using text merge

☐ Inactive

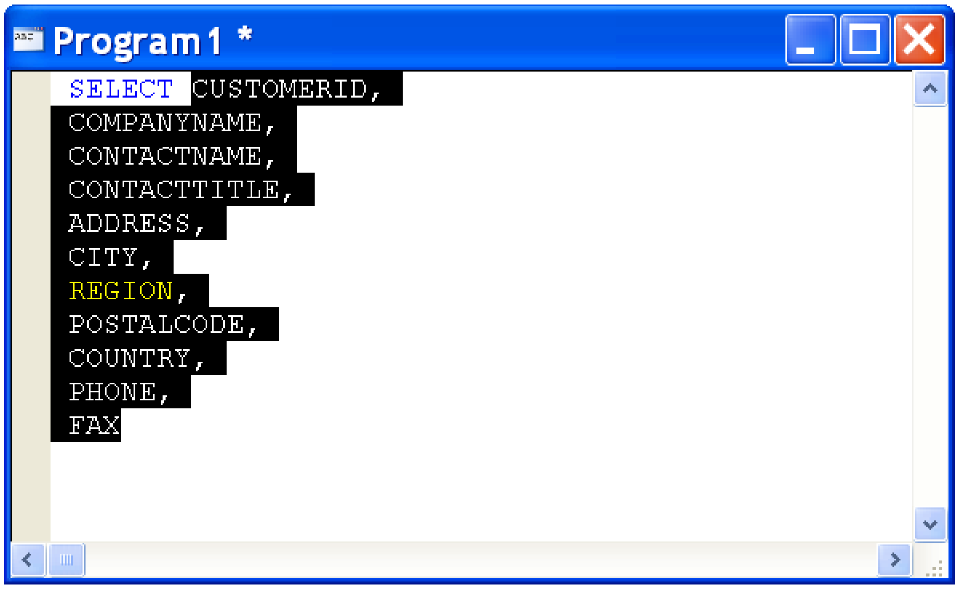
Figure 20. To set up a text scrap, specify a name and fill in the text to be inserted. If it uses textmerge, make sure to check the Evaluate checkbox.

Sometimes, you want to insert something more complex than you can specify with plain text plus textmerge. That's where Toolbox scripts come into play. A script is a block of VFP code. When you drag and drop it, the code runs and the value returned is inserted at the drop point. Listing 4 shows a script that creates a comma-separated list of fields for an open table. When you drag and drop it, it inserts that list into the code window, as in Figure 21.

Listing 4. This Toolbox script creates a list of fields from the table in the current work area.

```
* Drop a list of fields from
* the open table into the designated location.
* Limitations: table must be open and in current work area
*              table must be open in default data session

#DEFINE CRLF CHR(13) + CHR(10)

LOCAL aFieldList[1], cFieldString, nFieldCount

cFieldString = ""
```

```
nDataSession = SET("Datasession")
SET DATASESSION TO 1

IF USED() && is there a table open?

   nFieldCount = AFIELDS(aFieldList)

   FOR nField = 1 TO m.nFieldCount
      cFieldString = m.cFieldString + ;
                  aFieldList[m.nField, 1] + ", " + CRLF
   ENDFOR

   * Strip out final comma
   cFieldString = SUBSTR(m.cFieldString, 1, LEN(m.cFieldString) - 4) + CRLF
ENDIF

SET DATASESSION TO m.nDataSession

RETURN m.cFieldString
```



```
SELECT CUSTOMERID,
COMPANYNAME,
CONTACTNAME,
CONTACTTITLE,
ADDRESS,
CITY,
REGION,
POSTALCODE,
COUNTRY,
PHONE,
FAX
```

Figure 21. The script in Listing 4 inserts a comma-separated list of fields at the drop point.

The script has a few flaws. First, it doesn't take the current indentation into account. Toolbox scripts don't have any information about the drop location.

Second, it doesn't include semi-colons for line continuation. That was actually a design decision, since there are a number of ways this script could be used. If you envision using it only for code like that in Figure 21, you can modify the script to include the semi-colons.

The final issue is that the table needs to be open in the current work area in the default data session. My initial version of the script prompted the user to choose a table, but because they're based on drag and drop, scripts don't handle I/O well. It's best not to do any input or output in a script.

Apart from those limitations, a Toolbox script puts the whole functionality of VFP at your disposal. Creating a script is similar to creating a text scrap. Choose Add Item in the Customize Toolbox dialog and choose Script in the Add Item dialog. In the Item Properties dialog, specify a name for the script and type or paste the script into the Script editbox. Scripts have one more option than text scraps, though; you can specify an additional script to run after the drag has been completed. You might use this to do some clean-up work.

Reason 7: Multiple "clipboards"

There's actually an easier way to create text scraps. Just drag a block of text (whether it's code or something else) into the Toolbox and it becomes a text scrap. This is handy when you need to add several chunks of code in multiple places.

For example, I recently needed to go through a couple of very large classes and add code to save and restore the work area to every method that opened any tables or cursors. The code itself wasn't that complex, but there were two blocks to insert in each method, one at the top to save the work area and one at the bottom to restore it. So a regular cut-and-paste wasn't sufficient. The Toolbox provided an easy solution.

Reason 8: Mix and match visual and code classes

While it's generally best to subclass visual controls using the Class Designer, and reserve code for non-visual classes, sometimes there's a reason to design a particular class in code. More importantly, sometimes you need to include non-visual classes in forms and other classes. For example, you may have a subclass of Custom that provides behavior that you need on a form. If you create the subclass with the Class Designer, you can drop it onto the form, no matter which of the tools you use.

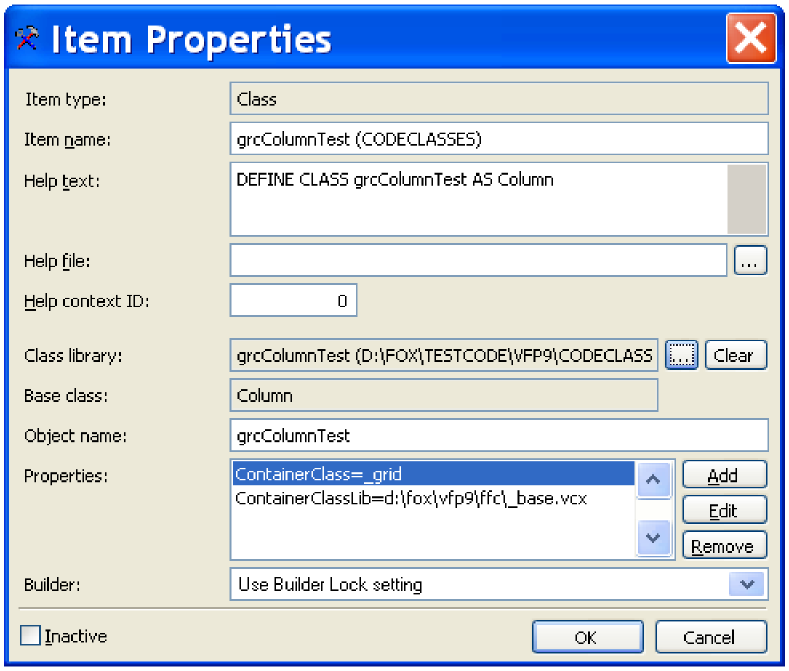
However, if you create the subclass in code (that is, in a PRG), the Form Controls toolbar and the Project Manager don't offer a way to drop it onto a form or another class. For that, you need to use either the Class Browser or the Toolbox.

The Toolbox lets you treat classes stored in a PRG pretty much the same as those stored in a VCX. You can drag and drop them onto forms and classes, and you can open them for editing right from the Toolbox.

An aside: adding containers and their contents

One of the Toolbox's tricks is that when you drop a control that has to be in a particular type of container (like an option button or page), it automatically adds the container as well. This applies to classes in a PRG, as well; where it's particularly useful is with grid columns, which can't be subclassed visually. When you drop a grid column class onto a form, the Toolbox adds a grid with a single column based on the class you dropped.

You can control the class used for the container by setting two special properties in the control's Item Properties dialog. The ContainerClass and ContainerClassLib properties exist only in this context, but when they're populated, dropping the contained control adds a container of the specified class. Figure 22 shows Item Properties for a column subclass, with these properties set to use the _grid class from the FFC's _base.vcx.



Item type: Class
Item name: grcColumnTest (CODECLASSES)
Help text: DEFINE CLASS grcColumnTest AS Column
Help file:
Help context ID: 0
Class library: grcColumnTest (D:\FOX\TESTCODE\VFP9\CODECLASS
Base class: Column
Object name: grcColumnTest
Properties: ContainerClass=_grid
ContainerClassLib=d:\fox\vfp9\ffc_base.vcx
Builder: Use Builder Lock setting
Inactive

Figure 22. You can specify what container class is used when a contained object is dropped by setting ContainerClass and ContainerClassLib..

Extending the Toolbox

By now, you should have a pretty good idea of why the Toolbox should become part of your regular toolkit. But with a little work, you can get even more out of this tool.

Like many of the tools that come with VFP, the Toolbox's source code is installed with VFP. (Source for all of what are called the Xbase tools is in XSource.ZIP in the Tools\XSource directory of your VFP installation.) So if you want to modify its behavior, you can change the code.

However, also like a lot of VFP tools, the Toolbox was designed to allow you to extend it without modifying its code. You can make some changes by adding records to tables, and others by writing a little code and making it available.

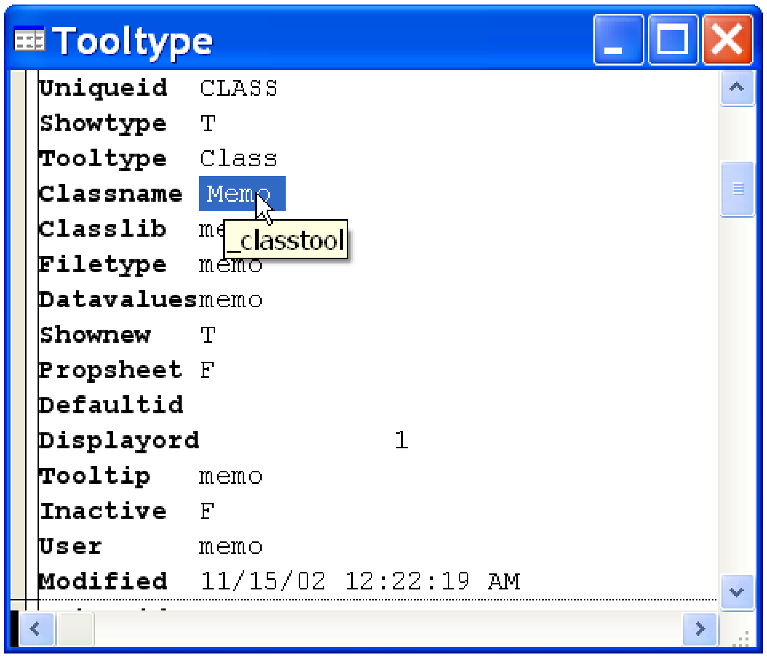
The remaining reasons to use the Toolbox also demonstrate ways to extend it, using a combination of the underlying tables, and code you write.

Reason 9: Store whatever you want

Although the Toolbox includes six types of items, and the file type is pretty broad, you might have other kinds of things you want to include in the Toolbox with their own behaviors. It turns out that adding new item types is fairly simple.

One of the projects I'm working on uses a Wiki and an online bug tracking system. While I do have shortcuts in my browser for each of those sites, having them accessible from within VFP seems pretty useful. You can use a File type item for a link, but getting it set up right is a little tricky. So I decide to add a special Link type.

The Toolbox stores the information about how to deal with various item types in a table called ToolType.DBF in the Toolbox folder under the VFP home directory. By default, it contains records for the built-in category and item types. It also contains records for VFP-specific file types to indicate how they should be handled when added as file items. Each record specifies a class that implements the various Toolbox operations for that type. Figure 23 shows the record in the ToolType table for the Class item type. It indicates that processing is handled by the _classtool class (which is contained in _Toolbox.VCX, the default location for such classes).

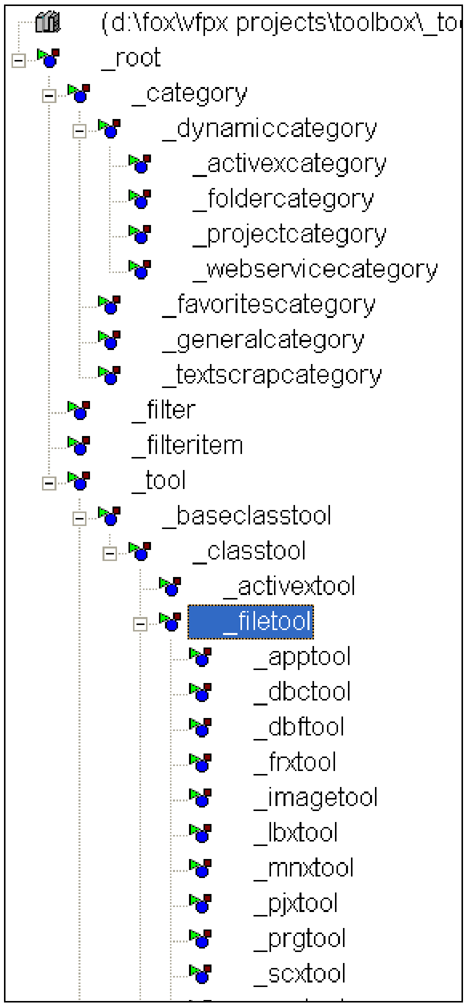


```
Uniqueid   CLASS
Showtype   T
Tooltype   Class
Classname  Memo
Classlib   me _classtool
Filetype   memo
Datavaluesmemo
Shownew    T
Propsheet  F
Defaultid
Displayord           1
Tooltip    memo
Inactive   F
User       memo
Modified   11/15/02 12:22:19 AM
```

Figure 23. The ToolType table contains a record for each type of item to indicate how it should managed.

To add a new item type, we need to add a record to ToolType and create a class to handle it. Although the source code for the Toolbox is provided, it's best if we can make this kind of change without modifying the source code. So rather than adding our link handling class to _Toolbox.VCX, we'll put it into a new class library (ToolboxExtension.VCX) in the same folder as the ToolType table.

Since the Toolbox already knows what to do with a link (open the page in the default browser) if we specify a link for a File item, we'll base the new class on the one used for files (called _FileTool). As Figure 24 shows, _FileTool comes from a significant hierarchy. All of the "tools" in the toolbox (including categories) derive from the _root class; in fact, much of the Toolbox's functionality is in that one class. Below that, things are divided into _category and _tool, and then subclassed from there. _FileTool handles all the functionality common to the File data item and is subclassed for functionality specific to a type of file.



```
(d:\fox\vfpx projects\toolbox\_to
_root
    _category
        _dynamiccategory
            _activexcategory
            _foldercategory
            _projectcategory
            _webservicecategory
        _favoritescategory
        _generalcategory
        _textscrapcategory
    _filter
    _filteritem
    _tool
        _baseclasstool
            _classtool
                _activextool
                _filetool
                    _apptool
                    _dbctool
                    _dbftool
                    _frxtool
                    _imagetool
                    _lbxtool
                    _mnxtool
                    _pjxtool
                    _prgtool
                    _scxtool
```

Figure 24. The tool classes used in the Toolbox all derive from a conmon ancestor, _root.

The only thing our new class has to do differently is provide a way to add links easily. That comes down to modifying the Item Properties form, so that it provides a textbox to enter the URL rather than prompting for a filename.

It took some digging into the Toolbox code to figure out what had to change. It turns out that the dialog bases its contents on a collection called oDataCollection. The OnCreateDataValues method of _FileTool adds an item to the collection to handle the filename, specifying that it should use a control class called cfoxfilename. The line of code from _FileTool.OnCreateDataValues that sets this up is shown in Listing 5.

Listing 5. This line adds the controls for the filename to the Item Properties dialog for a File item.

```
THIS.AddDataValue("filename", '', DATAVALUE_FILENAME_LOC, ;
                  '', .F., "cfoxfilename", '')
```

For our link-handling class, we want an ordinary textbox instead of the textbox and button combination that cfoxfilename specifies. That class comes from ToolboxCtrls.VCX; in the same class library, there's a regular textbox class, called cfoxtextbox. We'll use that one instead for our link class.

So, we need to create a subclass of _FileTool and call it _LinkTool. Then, in its OnCreateDataValues methods, we want to replace the cfoxfilename control with a cfoxtextbox. Listing 6 shows the code that does the trick. Rather than replacing the code in _FileTool (which does some other things as well), we simply modify the member of oDataCollection to have the value we want. This code also modifies the caption used next to the text. You'll also need to add the line in Listing 7 to the Toolbox.H file.

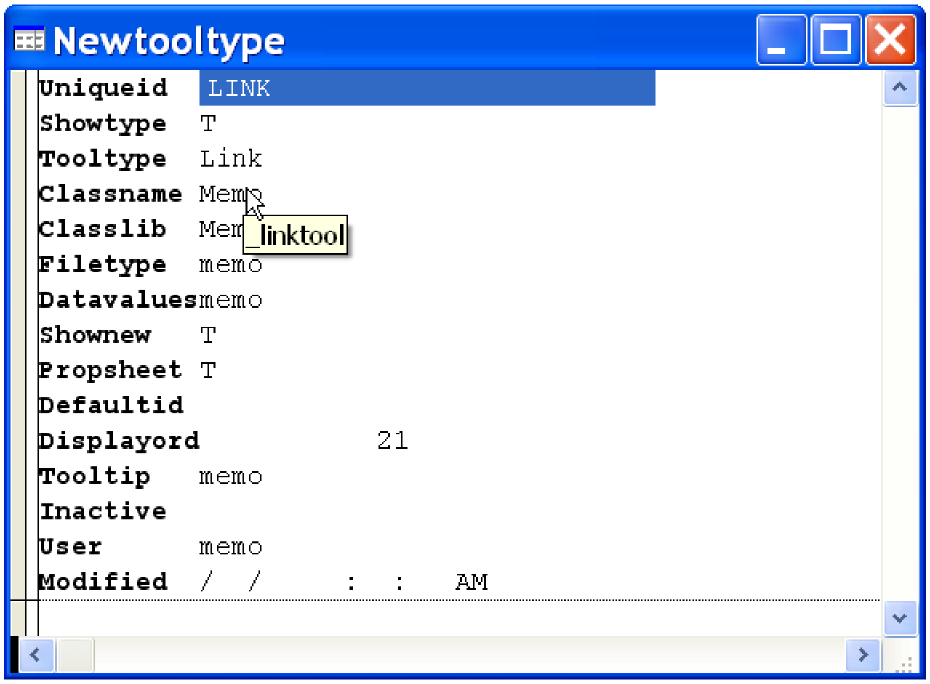
Listing 6. The new _LinkTool class has code in only one method, OnCreateDataValues.

```
#include "toolbox.h"

DODEFAULT()

* custom to the _linktool, modify the info for the filename item
WITH This.oDatacollection("FILENAME")
   .DataCaption = DATAVALUE_LINK_LOC
   .EditClass = "cFoxTextBox"
ENDWITH
```

Listing 7. Add this line to Toolbox.H to make the code in Listing 7 work.

```
#DEFINE DATAVALUE_LINK_LOC                "Link"
```

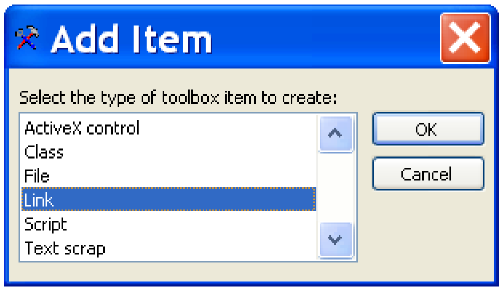
With the code done, all we have to do is add a record to ToolType.DBF to make the whole thing work. Figure 25 shows data for this record. Note that both Classname and ClassLib have data here—they point to the new _LinkTool class in ToolboxExtensions.VCX. (Unfortunately, I've found no way to do this with a relative path in ClassLib, so ClassLib points to the exact folder and file where I've placed the new library.) Setting ShowNew to .T. ensures that this type will show up in the Add Item dialog. Setting PropSheet to .T. causes the Item Properties dialog to appear as part of the process of adding an item.



```
Uniqueid   LINK
Showtype   T
Tooltype   Link
Classname  Memo
Classlib   Memo
Filetype   memo
Datavaluesmemo
Shownew    T
Propsheet  T
Defaultid
Displayord           21
Tooltip    memo
Inactive
User       memo
Modified   / /     : :    AM
```

Figure 25. Add this record to the ToolType table to set up the Link tool.

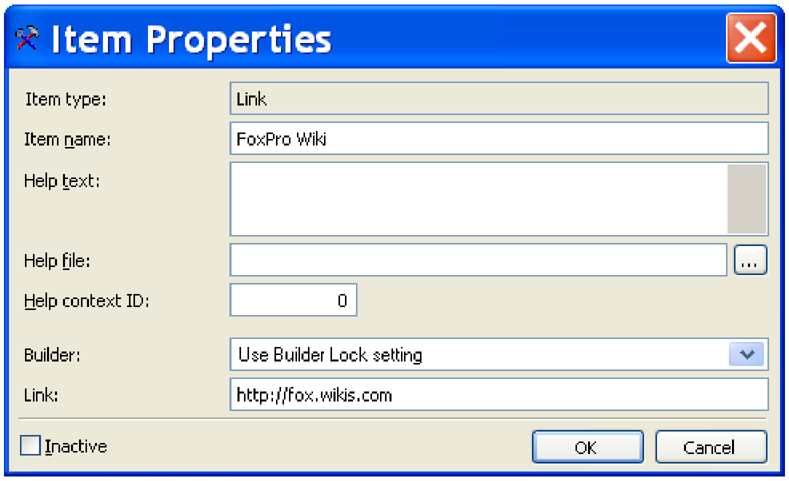
Once you've made these changes, when you choose Add Item from the Customize Toolbox dialog, the Add Item dialog includes the Link type (as shown in Figure 26). When you choose the Link type, the Item Properties dialog appears, with a textbox for entering the link (Figure 27). When you've added a link, you can click on it to open the page in your browser; the Toolbox is even smart enough to use a link icon for it.



Select the type of toolbox item to create:

- ActiveX control
- Class
- File
- Link
- Script
- Text scrap

Figure 26. After adding the record in Figure 25 to the ToolType table, the Add Item dialog includes the Link type.



Item type: Link

Item name: FoxPro Wiki

Help text:

Help file:

Help context ID: 0

Builder: Use Builder Lock setting

Link: http://fox.wikis.com

☐ Inactive

Figure 27. The Item Properties dialog for the new Link type includes a textbox to enter the URL.

Reason 10: Add your own menu items

Now that we have an item for adding links to the Toolbox, it would be nice if we could do it without having to open the Customize Toolbox dialog. It turns out that the Toolbox supports an add-in mechanism that lets you add items to the shortcut menus.

Add-ins are stored in the main Toolbox data table, Toolbox.DBF, which is stored by default in your user directory (the one HOME(7) points to). This table contains one record for each category and one record for each item in the Toolbox. By default, there are no add-ins, but you can set them up.

An add-in for a shortcut menu item needs only a few fields filled in: UniqueID, ShowType, ToolName and ToolData.

UniqueID is the item's primary key. The Toolbox uses a two-part naming scheme for UniqueID. All categories have UniqueID values in the form "Microsoft.CategoryName." The built-in items have UniqueID values in the form "CategoryName.ItemName." Items you add through the Toolbox's interface have UniqueID value in the form "User.SYS2015value." I recommend that when you add items manually, you use a similar format with the first part identifying you (or your company) and the second part identifying the item. So, for the Add link menu item, I'll use "Granor.AddLink" as the UniqueID.

ShowType indicates the type of item the record represents. It contains "C" for category, "T" for "tool item," and so forth. For an add-in, use "A".

ToolName is the name used for the item in the Toolbox. For a menu item, put the text you want in the menu. For the Add link menu item, I'll put "Add link" of course.

Finally, ToolData is the field that provides functionality. For an add-in, you put code to run when the add-in is chosen. For the Add link menu item, I modified code found from the

CreateToolItem method of the ToolboxEngine class; the modified code is shown in Listing 8.

Listing 8. This code implements the Add link menu item. It's stored in the ToolData memo field of the add-in's record in Toolbox.DBF.

```
LPARAMETERS oCurrentItem

LOCAL oEngine, oToolItem, oCategory, oToolType, lSuccess

oEngine = oCurrentItem.oEngine

oCategory = oEngine.CurrentCategory

oToolType = oEngine.GetTooltypeRec("LINK")
IF NOT ISNULL(m.oToolType)
   m.lShowPropertySheet = m.oToolType.PropSheet
   m.cClassName = m.oToolType.ClassName
   m.cClassLib  = m.oToolType.ClassLib
   IF EMPTY(m.cClassLib)
      m.cClassLib = oEngine.DefaultClassLib
   ENDIF

   TRY
      m.oToolItem = NEWOBJECT(m.cClassName, m.cClassLib)
   CATCH TO oException
      MESSAGEBOX(oException.Message + CHR(10) + CHR(10) + m.cClassName + ;
                 "(" + m.cClassLib + ")", MB_ICONEXCLAMATION, TOOLBOX_LOC)
   ENDTRY

   IF VARTYPE(m.oToolItem) == 'O'
      WITH m.oToolItem
         .oEngine    = m.oEngine
         .ToolTypeID = m.oToolType.UniqueID
         .ToolType   = m.oToolType.ToolType
         .ClassName  = m.cClassName
         .ClassLib   = IIF(m.cClassLib == oEngine.DefaultClassLib, '', m.cClassLib)
      ENDWITH

      IF m.lShowPropertySheet
         IF !oToolItem.Properties(.T.)
            m.oToolItem = .NULL.
         ENDIF
      ENDIF

      IF !ISNULL(m.oToolItem)
         WITH m.oToolItem
            .ParentID = oCategory.UniqueID
         ENDWITH

         lSuccess = oEngine.NewItem(m.oToolItem)
      ENDIF
   ENDIF

ENDIF

RETURN m.lSuccess
```

Once you've added the record to the Toolbox table and reopened the Toolbox, you can see the Add Link menu item when you right-click over an item or category. Unfortunately, it doesn't appear when you right-click over an empty space; changing that behavior would require changes to the core Toolbox code.

If you'd like to have a shortcut menu item for adding text scraps, you can do it the same way. Just change the UniqueId and ToolName fields, and modify the code for ToolData to use the string "TEXTSCRAP" instead of "LINK" in the call to GetToolTypeRec.

Adding a shortcut menu item to add files is different, though. When you add a file, you want the GetFile() dialog to appear so you can point to the file rather than the Item Properties dialog. I found the right code in the AddTool method of the form ToolboxCustomize and modified it as in Listing 9.

Listing 9. Adding a file is actually easier than adding other items. Put this code in ToolData for a shortcut menu item to add files.

```
LPARAMETERS oCurrentItem

LOCAL oEngine, oToolItem, oCategory, oToolType, lSuccess

oEngine = oCurrentItem.oEngine

oCategory = oEngine.CurrentCategory

m.cFilename = GETFILE()
IF !EMPTY(m.cFilename)
   m.lSuccess = oEngine.CreateToolsFromFile(oCategory.UniqueID, m.cFilename, .T.)
ENDIF

RETURN m.lSuccess
```

Although it's not relevant for these menu items, you can specify that a particular shortcut menu item appears only for a particular item type. To do so, put the item type in the ToolTypeID column.

The downloads for this session include AddMenuItems.PRG, which adds the necessary records to Toolbox.DBF to add these three menu items.

Reason 11: Create your own dynamic file sets

The section for Reason 4 shows you how to create a dynamic folder category that includes all the files in the specified folder. I mentioned there that you can also filter the category to include only files of a specified type.

The Toolbox comes with a built-in list of file types you can use to filter, such as Forms and Classes (which includes .SCX, .VCX, and .PRG files), Data Structures (.DBC, .DBF and .CDX files), or Images (.ANI, .BMP, .CUR, .DIB, .GIF, .ICO, .JPG). All told, there are 10 sets of file types included.

You can also filter on one or more extensions by just specifying them in the Category Properties dialog. In Figure 12, for example, you could limit the category to only .DOC files by specifying "*.DOC" in the File type combo.

While those options are sufficient for most needs, you might want to create your own lists. For example, for a Documentation category, you might want to include .DOC and .PDF files only. You might want to be able to filter a folder to show only video files, or only audio clips.

Unfortunately, to change this behavior, you do have to modify Toolbox code. My first attempt simply added the one additional filter I wanted. That required adding one line to the Init method of the cFoxFileTypesCombo class, and adding a definition to Toolbox.H. Not much work, but not very extensible either.

So I went back to the drawing board and realized that this was a good candidate for data-driving. I create a new table called ToolFileType with two columns: cFileDesc is the string that describes the set of file types, and cExts is the semicolon-separated list of file extensions. Table 1 shows the table contents with all the built-in groups, plus additional groups for sounds and documents.

Table 1. The new ToolFileType table contains the list of file type sets used to filter the contents of a dynamic folder category.

cFileDesc—Description	cExts—Extensions to include
All Files	*.*
Common	*.scx;*.vcx;*.prg;*.frx;*.lbx;*.mnx;*.dbc;*.qpr;*.h
All Source	*.scx;*.vcx;*.prg;*.frx;*.lbx;*.mnx;*.dbc;*.dbf;*.cdx;*.qpr;*.h
Forms and Classes	*.scx;*.vcx;*.prg
Reports and Labels	*.frx;*.lbx
Menus	*.mnx

cFileDesc—Description	cExts—Extensions to include
Programs	*.prg;*.h;*.qpr;*.mpr
Data Structures	*.dbc;*.dbf;*.cdx
Projects	*.pjx
Text	*.txt;*.xml;*.xsl;*.htm;*.html;*.log;*.asp;*.aspx
Images	*.ani;*.bmp;*.cur;*.dib;*.gif;*.ico;*.jpg
Sounds	*.wav;*.mp3
Documents	*.doc;*.pdf

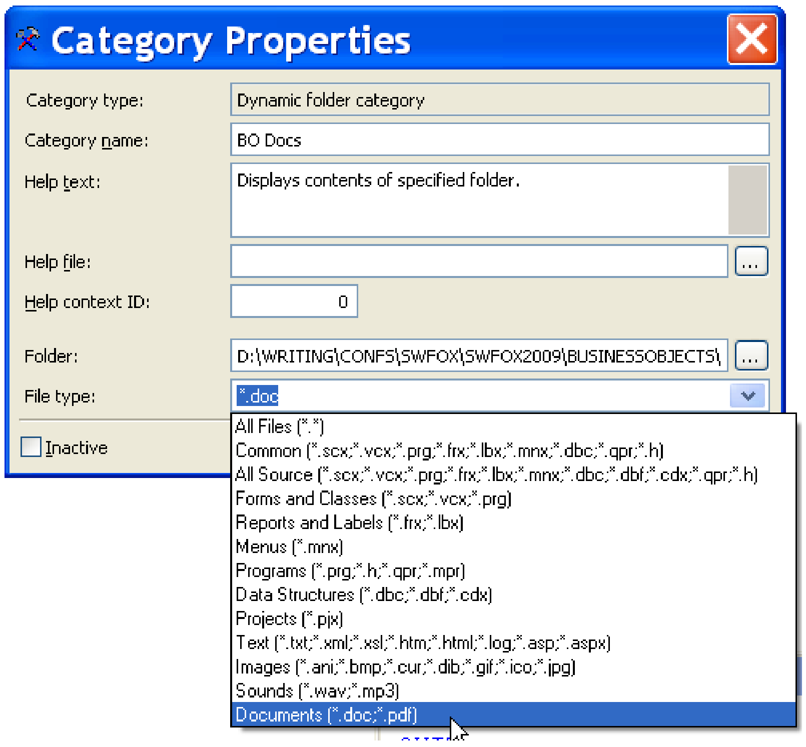
Of course, to use the new table, I still had to modify code. But this way, I could change it once. The place to make the change is still the Init of cFoxFileTypesCombo. I set it up so that if the new table can't be found, the original code is run. Listing 10 shows the new code.

Listing 10. Change the Init for the cboFoxFileTypesCombo to data-drive the list of file types.

```
#include "toolbox.h"
DODEFAULT()

* Modified 31-March-2009 by TEG
* If we have a table to list this, use that instead

IF FILE("ToolFileType.DBF")
   USE ToolFileType IN 0 ALIAS __ToolFileType
   SELECT __ToolFileType
   SCAN
      This.AddItem(ALLTRIM(cFileDesc) + " (" + ALLTRIM(cExts) + ")")
   ENDSCAN
   USE IN SELECT("__ToolFileType")
ELSE
   THIS.AddItem(FILETYPE_ALL_LOC)
   THIS.AddItem(FILETYPE_COMMON_LOC)
   THIS.AddItem(FILETYPE_SOURCE_LOC)
   THIS.AddItem(FILETYPE_FORMS_LOC)
   THIS.AddItem(FILETYPE_REPORTS_LOC)
   THIS.AddItem(FILETYPE_MENUS_LOC)
   THIS.AddItem(FILETYPE_PROGRAMS_LOC)
   THIS.AddItem(FILETYPE_DATA_LOC)
   THIS.AddItem(FILETYPE_PROJECTS_LOC)
   THIS.AddItem(FILETYPE_TEXT_LOC)
ENDIF
```

To use the new code, you have to rebuild the Toolbox application and use the new version. You have two options when you do this. You can either overwrite Toolbox.APP in the VFP home directory, or you can change _Toolbox so that it points to your new version. If you choose to put your version in the home directory, it's a good idea to make a back-up copy of the shipping version first.

Figure 28 shows the list of file types for a dynamic folder category, using the modified Toolbox with ToolFileType.DBF as specified in Table 1.


Category Properties

Category type:	Dynamic folder category
Category name:	BO Docs
Help text:	Displays contents of specified folder.
Help file:	
Help context ID:	0
Folder:	D:\WRITING\CONFS\SWFOX\SWFOX2009\BUSINESSOBJECTS\
File type:	*.doc

Inactive

All Files (*.*)
Common (*.scx;*.vcx;*.prg;*.frx;*.lbx;*.mnx;*.dbc;*.qpr;*.h)
All Source (*.scx;*.vcx;*.prg;*.frx;*.lbx;*.mnx;*.dbc;*.dbf;*.cdx;*.qpr;*.h)
Forms and Classes (*.scx;*.vcx;*.prg)
Reports and Labels (*.frx;*.lbx)
Menus (*.mnx)
Programs (*.prg;*.h;*.qpr;*.mpr)
Data Structures (*.dbc;*.dbf;*.cdx)
Projects (*.pjx)
Text (*.txt;*.xml;*.xsl;*.htm;*.html;*.log;*.asp;*.aspx)
Images (*.ani;*.bmp;*.cur;*.dib;*.gif;*.ico;*.jpg)
Sounds (*.wav;*.mp3)
Documents (*.doc;*.pdf)

Figure 28. With the updated Toolbox, you can choose Sounds or Documents as file types for a dynamic folder category. Add your own sets of extensions to the new ToolFileType table to specify types of files you deal with.

The source code for the updated Toolbox, as well as the new table, are included in the downloads for this session.

Reason 12: Handle moved class libraries

One problem when using the Toolbox is that if you move a class library for some reason, the Toolbox items that refer to its classes become incorrect. That is, the Toolbox's meta-data points to the folder where a class was stored when you added the class library. If you move the class library to another folder, you have to update that meta-data. While you can do so manually, it's not hard to create a Toolbox add-in to do the job.

As with the add-ins described in Reason 10, you can create this add-in by adding a record to Toolbox.DBF. I set UniqueID to "Granor.ChangeFolder" and ToolName to "Change category folder" for this one. ShowType is still "A" for "add-in." The code that goes in

ToolData is quite different than the code for adding the various new item types discussed earlier; it's shown in Listing 11.

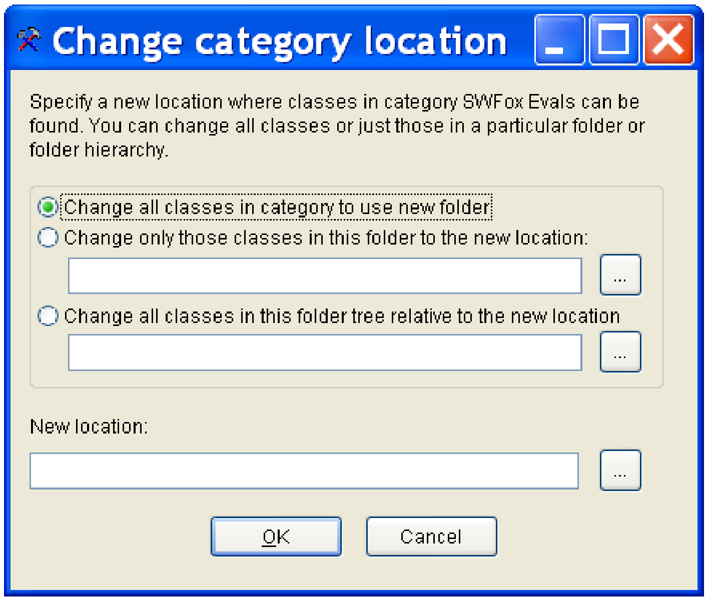
Listing 11. This code lets you change the folder for the classes in a particular category. It handles three different cases.

```
LPARAMETERS oCurrentItem

LOCAL oEngine, oTools, nItem, oTool, cCategory

oEngine = oCurrentItem.oEngine

IF UPPER(oCurrentItem.ClassType) = "CATEGORY"
   oTools = oEngine.GetToolsInCategory(oCurrentItem.UniqueID)
   cCategory = oCurrentItem.ToolName
ELSE
   oTools = oEngine.GetToolsInCategory(oEngine.CurrentCategory.UniqueID)
   cCategory = oEngine.CurrentCategory.ToolName
ENDIF

* Find out what the user wants to do
LOCAL oResult
DO FORM ChangeCategoryLocation WITH m.cCategory TO oResult

IF oResult.nAction <> 0 && 0 = user cancelled

   LOCAL cClassLib, lChangeFolder, cCurrentFolder

   FOR nItem = 1 TO oTools.COUNT
      oTool = oEngine.GetToolObject(oTools[m.nItem])
      IF UPPER(oTool.ClassType) = "CLASS"
         cClassLib = oTool.GetDataValue("ClassLib")
         cCurrentFolder = UPPER(JUSTPATH(m.cClassLib))

         lChangeFolder = .F.

         DO CASE
         CASE oResult.nAction = 1 && Change all
            lChangeFolder = .T.

         CASE oResult.nAction = 2 && Change only one folder
            IF m.cCurrentFolder = UPPER(oResult.cSourceFolder)
               lChangeFolder = .T.
            ENDIF

         CASE oResult.nAction = 3 && Change entire tree
            IF LEFT(m.cCurrentFolder, LEN(oResult.cSourceFolder)) == ;
              UPPER(oResult.cSourceFolder)
               lChangeFolder = .T.
            ENDIF
         OTHERWISE
            * No other cases
         ENDCASE
         IF m.lChangeFolder
            IF oResult.nAction = 1
               cNewClassLib = FORCEPATH(m.cClassLib, oResult.cDestFolder)
            ELSE
               cNewClassLib = STRTRAN(m.cClassLib, oResult.cSourceFolder, ;
                                      oResult.cDestFolder, 1, 1, 3)
            ENDIF
            oTool.SetDataValue("ClassLib", m.cNewClassLib)
            oEngine.SaveToolItem(m.oTool)
         ENDIF
      ENDIF
   ENDFOR

   oEngine.RefreshCategory()
ENDIF

RETURN
```

The code has three sections. The first part figures out what's going on. It gets a reference to the Toolbox engine. Then it figures out whether it was called from an item or a category. In either case, it finds the name of the category (the one containing the item, if called from an item) and creates a collection of the items in that category.

Next, a form (shown in Figure 29) is displayed to let you indicate what to do. There are three options:

- Change all classes in the category to point to the new folder.
- Change only classes in the category that are in a particular folder to point to the new folder.
- Change classes in the category in a particular folder hierarchy to point to the corresponding positions in a new folder hierarchy. (For example, if the original folder specified was C:\Fox\VFP8\Custom and the new folder was C:\Fox\VFP9\Custom, and you had classes from libraries in C:\Fox\VFP8\Custom\Controls and C:\Fox\VFP8\Custom\Forms in the category, after running this add-in, the classes would point to C:\Fox\VFP9\Custom\Controls and C:\Fox\VFP9\Custom\Forms, respectively.)


Change category location

Specify a new location where classes in category SWFox Evals can be found. You can change all classes or just those in a particular folder or folder hierarchy.

(●) Change all classes in category to use new folder
( ) Change only those classes in this folder to the new location:
( ) Change all classes in this folder tree relative to the new location

New location:

Figure 29. This form is called by the Change Category Folder add-in to let you specify what to do.

The form returns an object with three properties:

- nAction indicates which option button you choose. If you click Cancel, nAction is 0.
- cSourceFolder contains the original folder you specify for the second or third action types.
- cDestFolder contains the new folder you specify.

The final section of the code acts on your choices. The code loops through all the items in the category. If the item is a class, it figures out whether to change it, based on the action chosen, the current folder for the tool's class library, and the value of cSourceFolder. Then, if appropriate, the folder name is changed, and the tool information is saved.

I think a brief discussion of how I figured out what code to write is called for. Knowing that menu add-ins receive a parameter of oCurrentItem, the item where the context menu was fired, I started with the code in Listing 12 in the ToolData memo field. Then, I opened the Toolbox and selected my menu item. When execution stopped, I examined the oCurrentItem object and figured out what objects and properties are available. I then built the code gradually, using Suspend as needed to let me poke around some more.

Listing 12. This basic code in an add-in let me poke around to figure out what to do.

```
Lparameter oCurrentItem
Suspend
```

In addition to exploring the objects available in an add-in, I used the Toolbox source code as a reference. For example, initially, I couldn't figure out how to get a collection of the items in a category, but I knew there had to be a way. So I opened the Toolbox code (in this case, ToolboxEngine.PRG) and took a look at the method names using Document View in alphabetical order. Since the method I needed was called GetToolsInCategory, it wasn't hard to identify.

I used the same approach several times to figure out what Toolbox code I could take advantage of. One of the trickiest items to figure out was how to get the name of the class library for a class item. I could see it in the Toolbox table, but not in the ToolItem object. Digging around in the Toolbox code, I found the _Root class (the base object for all items) and was able to figure out that the relevant data was stored in oDataCollection. From there, it wasn't hard to discover the GetDataValue and SetDataValue methods.

Make the Toolbox work for you

By now, it should be clear that the Toolbox is both powerful and well-designed, so that not only can you easily make it work for you, but you can extend or modify it without too much difficulty. The harder I look at the Toolbox and its architecture, the more impressed I am.

There are a few resources available for the Toolbox. The chapter I wrote about it for *What's New in VFP 8* is available as a sample chapter at http://www.hentzenwerke.com/samplechapters/zsamplechapters.htm. Cindy Winegarden wrote a series of three articles about the Toolbox that appeared in FoxPro Advisor in January, April and June, 2003. Unfortunately, the online versions are available only to Databased Advisor subscribers. Finally, Beth Massi (who designed the Toolbox) wrote a white paper about extending it; it's at http://msdn.microsoft.com/en-us/library/ms965183.aspx.