Unit Testing VFP Applications

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

Unit testing is a very important part of application development and yet it seems that few VFP developers use this important technique. Unit testing allows you to confirm that code has the functionality expected of it, and is especially important when refactoring or making other changes to your code. This session introduces you to unit testing and the VFX FoxUnit project so you can be confident in your application's success before you deploy it.

Introduction

Unit testing seems to be a technique that isn't well known in the Visual FoxPro community. That may be partly because there are few tools available to assist with unit testing and partly because only a handful of presentations have been done and articles have been written on the topic. Hopefully after reading this document, you'll be inspired to start using unit testing as part of your development methodology.

Why unit testing is important

Have you ever had this happen?

- The customer asks for a change in the functionality of an existing application.
- You start sweating a bit because the code for that feature is pretty complex and you haven't looked at it in years (or it was written by another, long-departed developer). You're worried that modifying the code could break something.
- You update the code and test it informally. Everything seems to be working.
- You start sweating again as it comes time to deploy the updated application at the customer's site.
- You cross your fingers and close your eyes as the customer does a "smoke test," so-called from hardware engineers who plug something in and see if it smokes.

I've been through that scenario a hundred times. If you could learn a technique that would prevent the fear of changing existing code, wouldn't that be worth your time?

Unit testing is such a technique. Once you've built a stable of unit tests for an application, you can make a change and then run the tests to see if you broke anything. If the tests are complete enough—that is, covering as much of the code base as possible—your confidence in the code changes should be very high and you can minimize the fear of deploying to your customer.

Unit testing is useful for:

- Ensuring the code matches the specifications. If unit tests are written so they test that the code does all the things it's supposed to do, those tests will fail if the code isn't complete or missed doing some of what's expected.
- Finding bugs. Wouldn't you rather find a bug before you ship to your customer than have your customer find it? Since unit tests can hit every line of code and cover every possible scenario, they should uncover bugs at development time long before the customer sees the application.
- Adding or changing functionality. As I outlined in the scenario above, if you have a set of tests that cover the existing functionality of the code, you can run those tests to see if you broke anything after making changes. Of course, it's likely you'll have to add new tests or modify existing ones since the behavior of the application changed,

but those tests ensure the application continues to work as expected and become part of the existing set of tests for future use.

- Refactoring code. Sometimes, you realize that while the code works, there are performance issues, it could be more flexible (useful for future changes in functionality), or it just doesn't "smell" right (the design has problems). Changes you make to improve the performance, increase flexibility, or implement a better design don't change the functionality of the code, so you can run the existing unit tests as you make changes to ensure you aren't introducing bugs or changing the behavior from the specifications.
- Encouraging a better interface. A lot of times, the design for a class' interface seems reasonable until you actually start writing client code that uses the class. Often, I find little things like too many parameters or an awkward syntax for a method when I do that. When you write tests, you're essentially writing client code for the class, and typically sooner than you would "real" code. Many times I've tweaked my design after writing some tests and finding that I can improve how the class is used.
- Bringing new developers up to speed. Complex code can be difficult to follow but unit tests (at least good ones) are usually very simple. Sometimes it's easier to understand what a block of code is supposed to do by examining the tests to see what the expected behavior of the code is under differing circumstances.

Given all of these benefits, once you start doing unit testing, you'll wonder how you ever wrote code without it!

If you think about, you probably have already written some unit tests. You write a class, then you write a small PRG to instantiate that class and see if it works. The problem with that approach is that it isn't methodical and likely only covers a small part of the application.

Let's start looking at unit testing by discussing the principles of good unit testing.

Principles of good unit testing

In *The Art of Unit Testing*, Roy Osherove defines a unit test:

> A unit test is an automated piece of code that invokes the method or class being tested and then checks some assumptions about the logical behavior of that method or class. A unit test is almost always written using a unit-testing framework. It can be written easily and runs quickly. It's fully automated, trustworthy, readable, and maintainable.

He lists these properties of a good unit test:

- Automated and repeatable. You should be able to run a unit test without manually doing a bunch of setup, and the unit test should give the same results under the same conditions.

- Easy to implement. If a test is hard or takes more than a few minutes to write, it probably is trying to do too much. Also, you're much less likely to write unit tests if you perceive writing the tests as hard or time-consuming.
- Available for future use. Once a test is written, it should be available to be used over and over, not just as a one-shot deal.
- Anyone can run it. If another developer has a copy of the source code, they should be able to run the unit test without needing other components installed only on your system.
- Runs quickly. If the unit test takes too long to run, again it's likely trying to do too much. Also, tests that take too long to run will likely be avoided by busy developers who "don't have the time."

Our first tests

Let's create a simple class to process log files. It doesn't do much right now, just ensuring a valid filename is specified, but we want to make sure it does what it's supposed to. The class is shown in **Listing 1** (LogFileProcessor.PRG included with the sample files for this document).

Listing 1. A simple class we'll write tests for.

```
define class LogFileProcessor as Custom
   cErrorMessage = ''

   function IsValidFile(tcFileName)
      local llReturn
      do case
         case vartype(tcFileName) <> 'C' or empty(tcFileName)
            This.cErrorMessage = 'Invalid filename'
            llReturn = .F.
         case lower(justext(tcFileName)) <> 'log'
            This.cErrorMessage = 'Not a log file'
            llReturn = .F.
         case not file(tcFileName)
            This.cErrorMessage = 'File does not exist'
            llReturn = .F.
         otherwise
            llReturn = .T.
      endcase
      return llReturn
   endfunc
enddefine
```

Listing 2 (taken from LogFileProcessorTest.PRG) shows a set of tests for the IsValidFile method of LogFileProcessor. All of these tests pass.

Listing 2. Tests for the LogFileProcessor class.

```
* Instantiate the class.
```

```
loClass = newobject('LogFileProcessor', 'LogFileProcessor.prg')

* Test that IsValidFile handles no filename.

llResult = loClass.IsValidFile()
if llResult or loClass.cErrorMessage <> 'Invalid filename'
   messagebox('First test failed')
   return
endif
llResult = loClass.IsValidFile('')
if llResult or loClass.cErrorMessage <> 'Invalid filename'
   messagebox('Second test failed')
   return
endif

* Test that IsValidFile handles an invalid extension.

llResult = loClass.IsValidFile('x.txt')
if llResult or loClass.cErrorMessage <> 'Not a log file'
   messagebox('Third test failed')
   return
endif

* Test that IsValidFile handles a non-existent file.

llResult = loClass.IsValidFile('x.log')
if llResult or loClass.cErrorMessage <> 'File does not exist'
   messagebox('Fourth test failed')
   return
endif

* Test that IsValidFile works.

strtofile('', 'x.log')
llResult = loClass.IsValidFile('x.log')
if not llResult
   messagebox('Fifth test failed')
   return
endif
erase x.log
messagebox('All tests passed')
```

Now suppose a new requirement is added: the file cannot be empty. Here's a new CASE statement added to the code:

```
case empty(filetostr(tcFileName))
   This.cErrorMessage = 'File is empty'
   llReturn = .F.
```

Now the last test fails. We should modify that test to expect it to fail and add another test that passes by having some content in the file. Here are the updated tests:

```
* Test that IsValidFile handles an empty file.
```

```
strtofile('', 'x.log')
llResult = loClass.IsValidFile('x.log')
if llResult or loClass.cErrorMessage <> 'File is empty'
   messagebox('Fifth test failed')
   return
endif

* Test that IsValidFile works.

strtofile('some text', 'x.log')
llResult = loClass.IsValidFile('x.log')
if not llResult
   messagebox('Sixth test failed')
   return
endif
```

Note the good things about these tests:

- They completely test the functionality of the IsValidFile method.
- Each test is small, simple, and tests only one thing.

However, note the weaknesses of these tests:

- They use MESSAGEBOX() to display success or failure. That means the tests can't be automated because someone has to be there to respond to the dialogs.
- They use hard-coded strings to test the error messages. If you change the messages in IsValidFile, the tests break even though they correctly test the functionality of the method. However, the only way around that is to rewrite the class to use an error code or look up the string to use for a particular error somewhere, such as a table, and have the tests do the same.
- The code is in one long program without a clear distinction (other than comments) between tests. It isn't possible to run only some tests.

Better unit tests

Let's rewrite our tests to more closely conform to Roy's definition of a unit test; see **Listing 3**, taken from BetterLogFileProcessorTest.PRG. The improvements in this code are that each test is now independent and results are logged to a table rather than being displayed in dialogs. The main routine still runs all the tests but that's a minor issue. Now, if we have a collection of such test programs, a master test program could run them all and then analyze the TestResults table.

Listing 3. Our rewritten tests can be automated.

```
* Create a table to hold test results if necessary.

do case
   case used('TestResults')
   case file('TestResults.dbf')
```

```
      use TestResults in 0
   otherwise
      create table TestResults free (Test C(90), When T, Result L)
endcase

* Run the tests.

do Test1
do Test2
do Test3
do Test4
do Test5

* Display the results.

browse

* Helper method to log test results.

function LogResult(tcTest, tlResult)
insert into TestResults values (tcTest, datetime(), tlResult)

* Test that IsValidFile handles no filename.

function Test1
loClass  = newobject('LogFileProcessor', 'LogFileProcessor.prg')
llResult = not loClass.IsValidFile()
LogResult('IsValidFile handles no filename (no parameter)', llResult)
llResult = not loClass.IsValidFile('')
LogResult('IsValidFile handles no filename (empty parameter)', llResult)

* Test that IsValidFile handles an invalid extension.

function Test2
loClass  = newobject('LogFileProcessor', 'LogFileProcessor.prg')
llResult = not loClass.IsValidFile('x.txt')
LogResult('IsValidFile handles an invalid extension', llResult)

* Test that IsValidFile handles a non-existent file.

function Test3
loClass  = newobject('LogFileProcessor', 'LogFileProcessor.prg')
llResult = not loClass.IsValidFile('x.log')
LogResult('IsValidFile handles a non-existent file', llResult)

* Test that IsValidFile works.

function Test4
strtofile('some text', 'x.log')
loClass  = newobject('LogFileProcessor', 'LogFileProcessor.prg')
llResult = loClass.IsValidFile('x.log')
LogResult('IsValidFile works when all requirement met', llResult)
erase x.log

* Test that IsValidFile handles an empty file.
```

```
function Test5
strtofile('', 'x.log')
loClass  = newobject('LogFileProcessor', 'LogFileProcessor.prg')
llResult = not loClass.IsValidFile('x.log')
LogResult('IsValidFile handles an empty file', llResult)
erase x.log
```

Our tests still have a few weaknesses:

- Their names aren't very useful: they don't describe what each test is testing for.
- We have to modify the program to run tests individually.
- In this case, LogFileProcessor's Init method doesn't accept any parameters, so it's no big deal to instantiate the class in each function. However, if the class is changed to accept one or more parameters (such as the filename), each function has to be changed, duplicating code.
- We've built some infrastructure here (creating/opening the TestResults table and logging the results) that has to be reproduced in every test program.

Let's look at some solutions to these issues.

Naming tests

Tests should be named to tell you what they test. Here are the general rules of thumb I use:

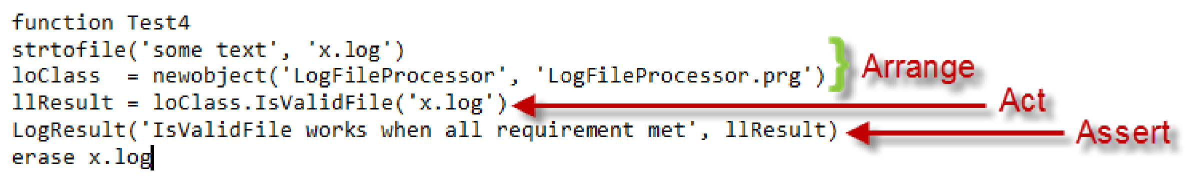
- The name of the test program is *ClassName*Test, where *ClassName* is the name of the class or program being tested. All of the tests for the class or program go into that test program so they're all in one place.
- The name of each test function is *MethodName_Scenario_ExpectedBehavior*, where *MethodName* is the name of the method being tested, *Scenario* is what behavior is being tested, and *ExpectedBehavior* is what we expect to happen. Here are some reasonable names for the tests in **Listing 3**: IsValidFile_NoFileName_ReturnsFalse, IsValidFile_InvalidExtension_ReturnsFalse, IsValidFile_NonExistentFile_ReturnsFalse, IsValidFile_EmptyFile_ReturnsFalse, and IsValidFile_ValidFile_ReturnsTrue.

Layout of a unit test

For consistency, the code in a unit test should do its tasks in this order:

- Arrange: set things up that the test requires.
- Act: invoke the method to be tested.
- Assert: test the results.

For example, Test4 in **Listing 3** uses this AAA style (**Figure 1**).



```
function Test4
strtofile('some text', 'x.log')
loClass  = newobject('LogFileProcessor', 'LogFileProcessor.prg')
llResult = loClass.IsValidFile('x.log')
LogResult('IsValidFile works when all requirement met', llResult)
erase x.log
```

Figure 1. Test4 shows the Arrange, Act, Assert (AAA) style of unit tests.

Although instantiating the class under test in each test function is fine, you may wish to refactor that out to a common helper method. That way, any arrange tasks common to all tests can be put in one place and if additional tasks are required because of changes to the class being tested, such as passing a parameter to Init or calling a Setup method, you only have to make those changes in one place.

Listing 4 (EvenBetterLogFileProcessorTest.PRG) has better names for the tests and refactors instantiation of the class under test to a helper method.

Listing 4. Our improved tests have better names and refactor class instantiation to a helper method.

```
* Create a table to hold test results if necessary.

do case
   case used('TestResults')
   case file('TestResults.dbf')
      use TestResults in 0
   otherwise
      create table TestResults free (Test C(90), When T, Result L)
endcase

* Run the tests.

do IsValidFile_NoFileName_ReturnsFalse
do IsValidFile_InvalidExtension_ReturnsFalse
do IsValidFile_NonExistentFile_ReturnsFalse
do IsValidFile_EmptyFile_ReturnsFalse
do IsValidFile_ValidFile_ReturnsTrue

* Display the results.

browse

* Helper method to log test results.

function LogResult(tcTest, tlResult)
insert into TestResults values (tcTest, datetime(), tlResult)

* Helper method to instantiate the class under test.

function CreateFileProcessor
loClass = newobject('LogFileProcessor', 'LogFileProcessor.prg')
return loClass

* Test functions.
```

```
function IsValidFile_NoFileName_ReturnsFalse
loClass  = CreateFileProcessor()
llResult = not loClass.IsValidFile()
LogResult('IsValidFile handles no filename (no parameter)', llResult)
llResult = not loClass.IsValidFile('')
LogResult(program(), llResult)

function IsValidFile_InvalidExtension_ReturnsFalse
loClass  = CreateFileProcessor()
llResult = not loClass.IsValidFile('x.txt')
LogResult(program(), llResult)

function IsValidFile_NonExistentFile_ReturnsFalse
loClass  = CreateFileProcessor()
llResult = not loClass.IsValidFile('x.log')
LogResult(program(), llResult)

function IsValidFile_EmptyFile_ReturnsFalse
strtofile('', 'x.log')
loClass  = CreateFileProcessor()
llResult = not loClass.IsValidFile('x.log')
LogResult(program(), llResult)
erase x.log

function IsValidFile_ValidFile_ReturnsTrue
strtofile('some text', 'x.log')
loClass  = CreateFileProcessor()
llResult = loClass.IsValidFile('x.log')
LogResult(program(), llResult)
erase x.log
```

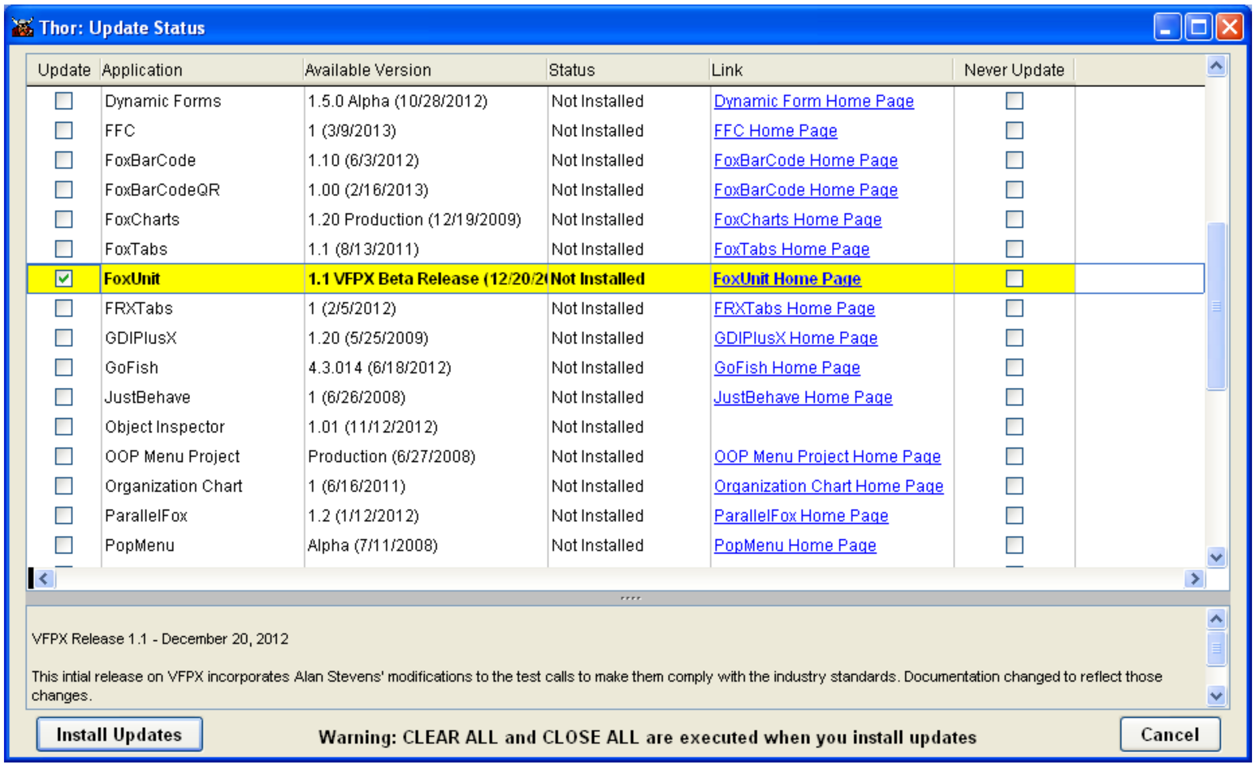
Using a testing framework: introducing FoxUnit

A testing framework helps automate testing in numerous ways:

- It provides an infrastructure for tests. Tests are typically methods within a test class subclassed from a framework class. This provides features such as assertions, which as we'll see provide both testing and logging capabilities, and built-in helper methods.
- It provides a UI to manage a suite of tests, provide one-click running of tests, and the ability to review test results, such as which tests were run, which ones failed, and why they failed.

Other development platforms, such as .Net and Java, have a wide assortment of testing frameworks available, including the very popular NUnit. In 2004, Jim Erwin of Visionpace, with assistance from other developers including Drew Speedie, created FoxUnit, a framework modeled somewhat on NUnit for creating and running VFP unit tests. Since then, FoxUnit has been released as a VFPX project (http://vfpx.codeplex.com), managed by Eric Selje.

The easiest way to install FoxUnit is if you use Thor, another VFPX project. From the Thor menu, choose Check for Updates, and in the dialog that appears, turn on the Update checkbox for FoxUnit, then click Install Updates; see **Figure 2**. After the installation is done, you run FoxUnit by choosing it from the Applications submenu in the Thor Tools menu. If you'd rather not use Thor, you can download the FoxUnit source from the Source Code page of VFPX.



Update	Application	Available Version	Status	Link	Never Update
☐	Dynamic Forms	1.5.0 Alpha (10/28/2012)	Not Installed	Dynamic Form Home Page	☐
☐	FFC	1 (3/9/2013)	Not Installed	FFC Home Page	☐
☐	FoxBarCode	1.10 (6/3/2012)	Not Installed	FoxBarCode Home Page	☐
☐	FoxBarCodeQR	1.00 (2/16/2013)	Not Installed	FoxBarCode Home Page	☐
☐	FoxCharts	1.20 Production (12/19/2009)	Not Installed	FoxCharts Home Page	☐
☐	FoxTabs	1.1 (8/13/2011)	Not Installed	FoxTabs Home Page	☐
☑	**FoxUnit**	**1.1 VFPX Beta Release (12/20/2**	**Not Installed**	**FoxUnit Home Page**	☐
☐	FRXTabs	1 (2/5/2012)	Not Installed	FRXTabs Home Page	☐
☐	GDIPlusX	1.20 (5/25/2009)	Not Installed	GDIPlusX Home Page	☐
☐	GoFish	4.3.014 (6/18/2012)	Not Installed	GoFish Home Page	☐
☐	JustBehave	1 (6/26/2008)	Not Installed	JustBehave Home Page	☐
☐	Object Inspector	1.01 (11/12/2012)	Not Installed		☐
☐	OOP Menu Project	Production (6/27/2008)	Not Installed	OOP Menu Project Home Page	☐
☐	Organization Chart	1 (6/16/2011)	Not Installed	Organization Chart Home Page	☐
☐	ParallelFox	1.2 (1/12/2012)	Not Installed	ParallelFox Home Page	☐
☐	PopMenu	Alpha (7/11/2008)	Not Installed	PopMenu Home Page	☐

VFPX Release 1.1 - December 20, 2012

This intial release on VFPX incorporates Alan Stevens' modifications to the test calls to make them comply with the industry standards. Documentation changed to reflect those changes.

Warning: CLEAR ALL and CLOSE ALL are executed when you install updates

Figure 2. It's easy to install FoxUnit using Thor.

The FoxUnit UI appears in **Figure 3**. We'll look at the features of the UI in more detail later, but for now, let's create some unit tests.

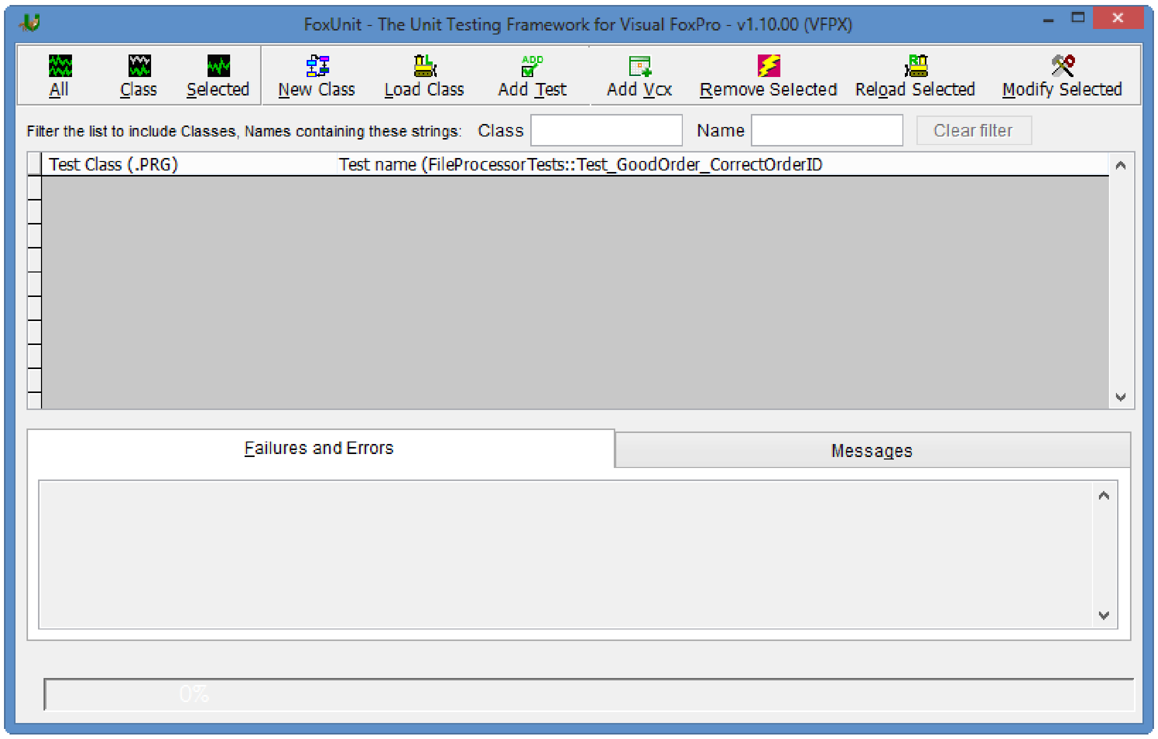
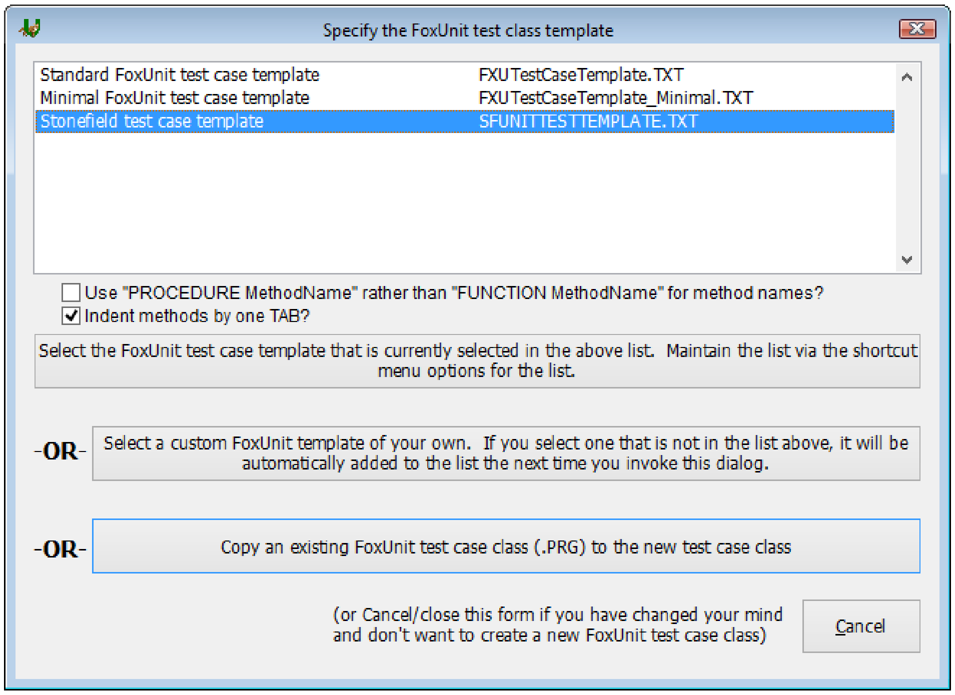

FoxUnit - The Unit Testing Framework for Visual FoxPro - v1.10.00 (VFPX)
All
Class
Selected
New Class
Load Class
Add Test
Add Vcx
Remove Selected
Reload Selected
Modify Selected
Filter the list to include Classes, Names containing these strings:
Class
Name
Clear filter
Test Class (.PRG)
Test name (FileProcessorTests::Test_GoodOrder_CorrectOrderID
Failures and Errors
Messages
0%

Figure 3. The FoxUnit user interface provides functions to create, manage, and run tests.

Click the New Class button to create a new test class. Specify the location of the tests (the default is the Test subdirectory of the current folder) and the filename for the PRG to be created. After you click OK, you're prompted for the type of template to use (see **Figure 4**). The choices are:

- The standard template. This template is stored in FXUTestCaseTemplate.txt, which you can modify as desired by right-clicking the item in the list and choosing Modify (note: currently, this gives an error if you installed FoxUnit using Thor because the folder containing the templates doesn't exist). This template includes lots of comments that help you to learn FoxUnit.
- The minimal template. This template, stored in stored in FXUTestCaseTemplate_Minimal.txt, only includes templates for the Setup and Teardown methods.
- A template of your own. This is just a PRG or text file used as the source for the new test class. You can add it to the list by right-clicking an existing item and choosing Add a Template or clicking the "Select a custom FoxUnit template" button. I've created a simple template called SFUnitTestTemplate.txt that's even more minimal than the minimal template but has a couple of features I use.
- Copy an existing test class.


Specify the FoxUnit test class template

Standard FoxUnit test case template	FXUTestCaseTemplate.TXT
Minimal FoxUnit test case template	FXUTestCaseTemplate_Minimal.TXT
Stonefield test case template	SFUNITTESTTEMPLATE.TXT

[ ] Use "PROCEDURE MethodName" rather than "FUNCTION MethodName" for method names?
[✓] Indent methods by one TAB?

Select the FoxUnit test case template that is currently selected in the above list. Maintain the list via the shortcut menu options for the list.

-OR- Select a custom FoxUnit template of your own. If you select one that is not in the list above, it will be automatically added to the list the next time you invoke this dialog.

-OR- Copy an existing FoxUnit test case class (.PRG) to the new test case class

(or Cancel/close this form if you have changed your mind and don't want to create a new FoxUnit test case class)

Cancel

Figure 4. Select the type of test class template you wish to use.

Listing 5 has the tests from **Listing 4** as FoxUnit tests.

Listing 5. The FoxUnit tests for the LogFileProcessor class.

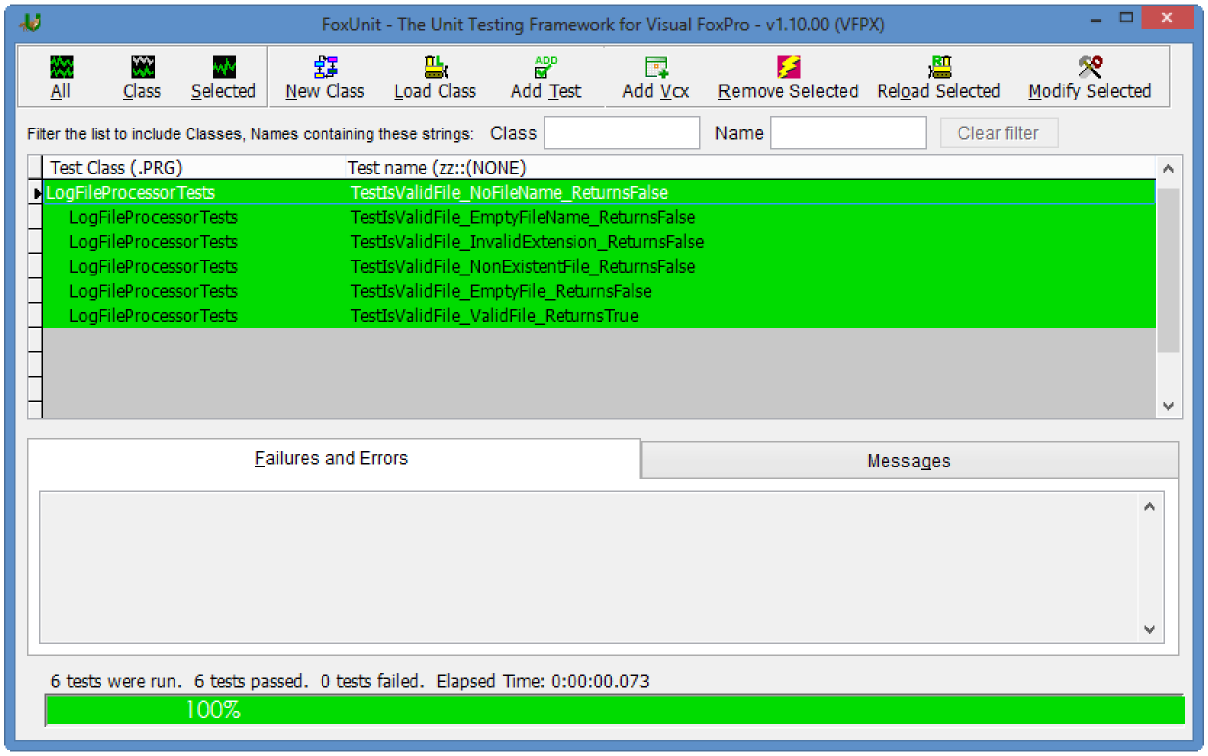
```
define class LogFileProcessorTests as FxuTestCase of FxuTestCase.prg
   #if .f.
      local this as LogFileProcessorTests of LogFileProcessorTests.PRG
   #endif

   icTestPrefix = 'Test'
   oObjectToTest = .NULL.

   function Setup
      This.oObjectToTest = newobject('LogFileProcessor', 'LogFileProcessor.prg')
   endfunc

   function TearDown
      erase x.log
   endfunc

   function TestIsValidFile_NoFileName_ReturnsFalse
      llResult = This.oObjectToTest.IsValidFile()
      This.AssertFalse(llResult)
   endfunc

   function TestIsValidFile_EmptyFileName_ReturnsFalse
      llResult = This.oObjectToTest.IsValidFile('')
      This.AssertFalse(llResult)
   endfunc
```

```
    function TestIsValidFile_InvalidExtension_ReturnsFalse
       llResult = This.oObjectToTest.IsValidFile('x.txt')
       This.AssertFalse(llResult)
    endfunc

    function TestIsValidFile_NonExistentFile_ReturnsFalse
       erase x.log
       llResult = This.oObjectToTest.IsValidFile('x.log')
       This.AssertFalse(llResult)
    endfunc

    function TestIsValidFile_EmptyFile_ReturnsFalse
       strtofile('', 'x.log')
       llResult = This.oObjectToTest.IsValidFile('x.log')
       This.AssertFalse(llResult)
    endfunc

    function TestIsValidFile_ValidFile_ReturnsTrue
       strtofile('some text', 'x.log')
       llResult = This.oObjectToTest.IsValidFile('x.log')
       This.AssertTrue(llResult)
    endfunc
enddefine
```

Here are some things to note about this code:

- All tests are included in a single class which is a subclass of FxuTestCase, one of the FoxUnit classes.
- The LOCAL statement inside the #IF block is there so IntelliSense works on the class.
- The built-in icTestPrefix property is set to "Test," which tells FoxUnit that only methods starting with "Test" are considered to be test methods. This allows you to add helper methods that test methods call without FoxUnit treating them like test methods.
- The Setup method, which runs at the start of each test in the class, instantiates LogFileProcessor into the custom oObjectToTest property. This allows test methods to simply use that property rather than calling a helper method like **Listing 4** did or instantiating the class themselves.
- The Teardown method, which runs after each test is done, erases x.log because some tests create that file; this means the test class cleans up after itself.
- Each test method calls the IsValidFile method of This.oObjectToTest with various parameters, and uses an assert method to indicate whether the test succeeded or failed.

Click the Class button in the FoxUnit toolbar to run all tests (the All button runs all tests in all classes and Selected runs just the test selected in the list). Each test is highlighted in green if it passed and red if not, and the results at the bottom display in green if all tests pass or red if at least one test failed. See **Figure 5**.

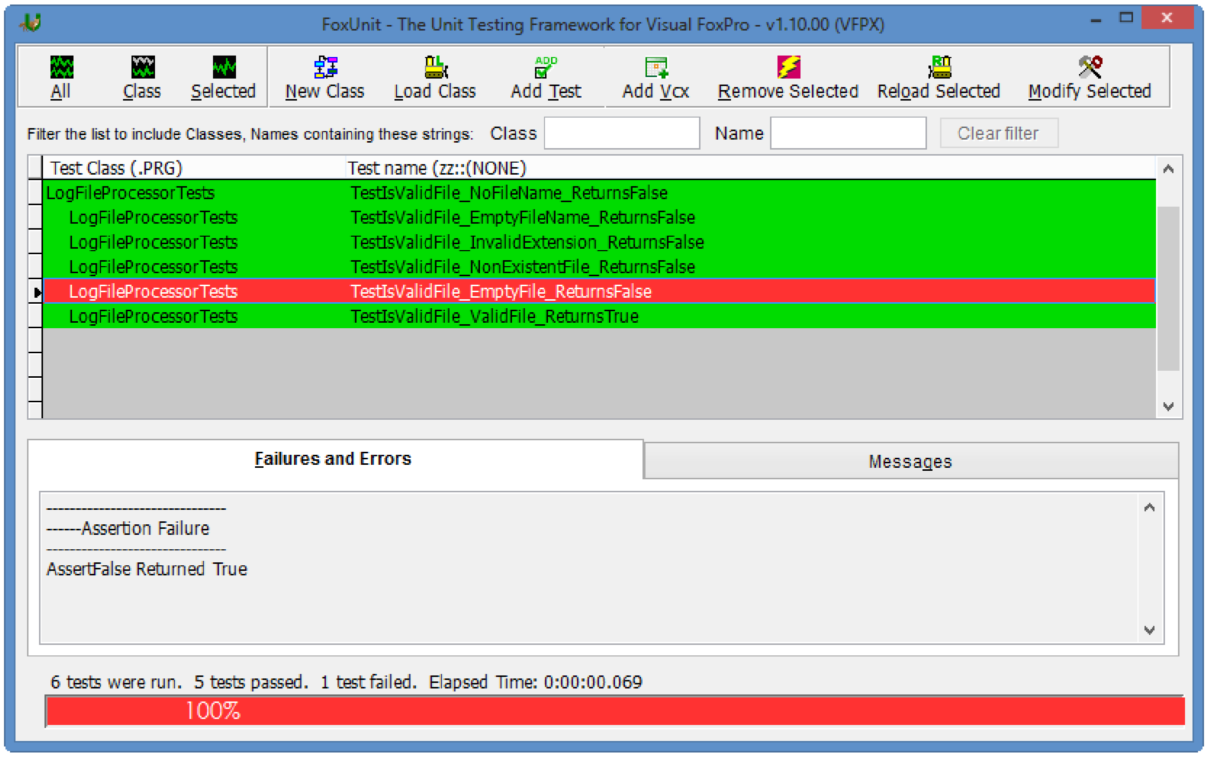


Test Class (.PRG)	Test name (zz::(NONE)
LogFileProcessorTests	TestIsValidFile_NoFileName_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_EmptyFileName_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_InvalidExtension_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_NonExistentFile_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_EmptyFile_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_ValidFile_ReturnsTrue

```
6 tests were run.  6 tests passed.  0 tests failed.  Elapsed Time: 0:00:00.073
100%
```

Figure 5. If all tests pass, FoxUnit shows green.

Figure 6 shows what FoxUnit looks like when one of the tests fails.



Test Class (.PRG)	Test name (zz::(NONE)
LogFileProcessorTests	TestIsValidFile_NoFileName_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_EmptyFileName_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_InvalidExtension_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_NonExistentFile_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_EmptyFile_ReturnsFalse
LogFileProcessorTests	TestIsValidFile_ValidFile_ReturnsTrue

```
-------------------------------
------Assertion Failure
-------------------------------
AssertFalse Returned True
```

6 tests were run. 5 tests passed. 1 test failed. Elapsed Time: 0:00:00.069

100%

Figure 6. FoxUnit displays failed tests in red.

FoxUnit features

In addition to the ones we've already discussed, FoxUnit has the following features:

- Its main form has Desktop set to .T. so you can move its window outside the VFP _screen.
- To load a test class into the list, click Load Class. The Load Class dialog (**Figure 7**) displays a list of PRGs in the default folder, but you can also select a different folder by clicking Browse. Turn on Test Cases Only to omit PRGs that don't contain FoxUnit tests. Turn on New Test Cases to only display PRGs that aren't already loaded. Select one or more PRGs and click Load to load them.

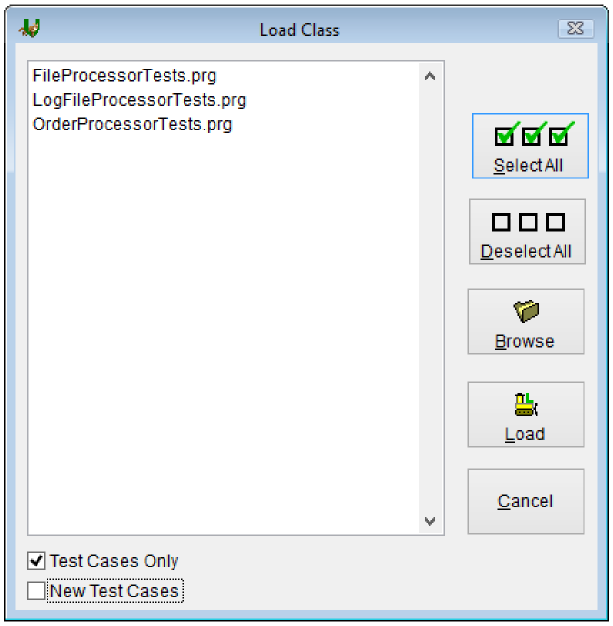
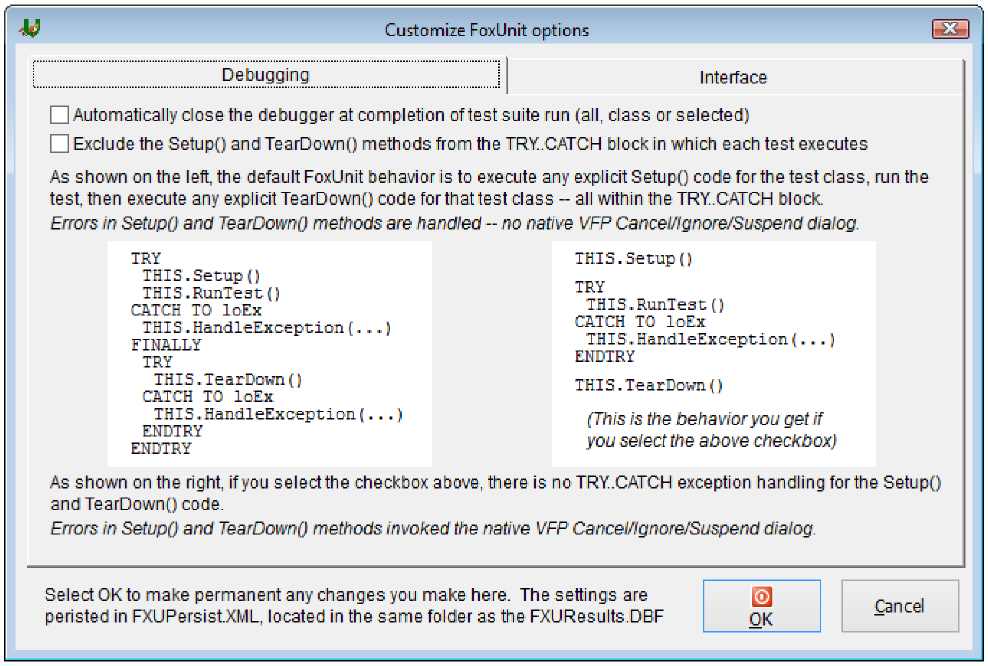


Figure 7. The Load Class dialog allows you to add additional test classes to the list.

- Click Add Test to add a test to the selected test class.
- Add VCX allows you to select a VCX and automatically generates test methods for each of the public methods in each of the classes in that VCX. These are generic tests so each needs to be modified to actually do the correct thing.
- Remove Selected removes the tests in the selected test class from the list, but doesn't delete them. Use Load Class to add them back to the list.
- Click Reload Selected to reload the selected test class in case you edited the PRG outside of FoxUnit.
- To edit a specific test, double-click it in the list or click the Modify Selected button.
- To filter the list of tests, enter strings used to filter the class and/or test names and press Tab. The Clear Filter button clears the filter.
- The Failures and Errors tab displays assertion failures (that is, the call to an assert method returned false) and errors that occurred during the last test run.
- You can change the font for the test list, the Failures and the Error and Message tabs by right-clicking and choosing Font. Choose Reset Font to Default to reset the font.
- The Acknowledgements and License functions in the shortcut menu display the appropriate text.
- **Figure 8** and **Figure 9** show the dialog displayed by the Options function in the shortcut menu. These options are well-described.



☐ Automatically close the debugger at completion of test suite run (all, class or selected)
☐ Exclude the Setup() and TearDown() methods from the TRY..CATCH block in which each test executes

As shown on the left, the default FoxUnit behavior is to execute any explicit Setup() code for the test class, run the test, then execute any explicit TearDown() code for that test class -- all within the TRY..CATCH block.
Errors in Setup() and TearDown() methods are handled -- no native VFP Cancel/Ignore/Suspend dialog.

```
TRY
 THIS.Setup()
 THIS.RunTest()
CATCH TO loEx
 THIS.HandleException(...)
FINALLY
 TRY
  THIS.TearDown()
 CATCH TO loEx
  THIS.HandleException(...)
 ENDTRY
ENDTRY
```

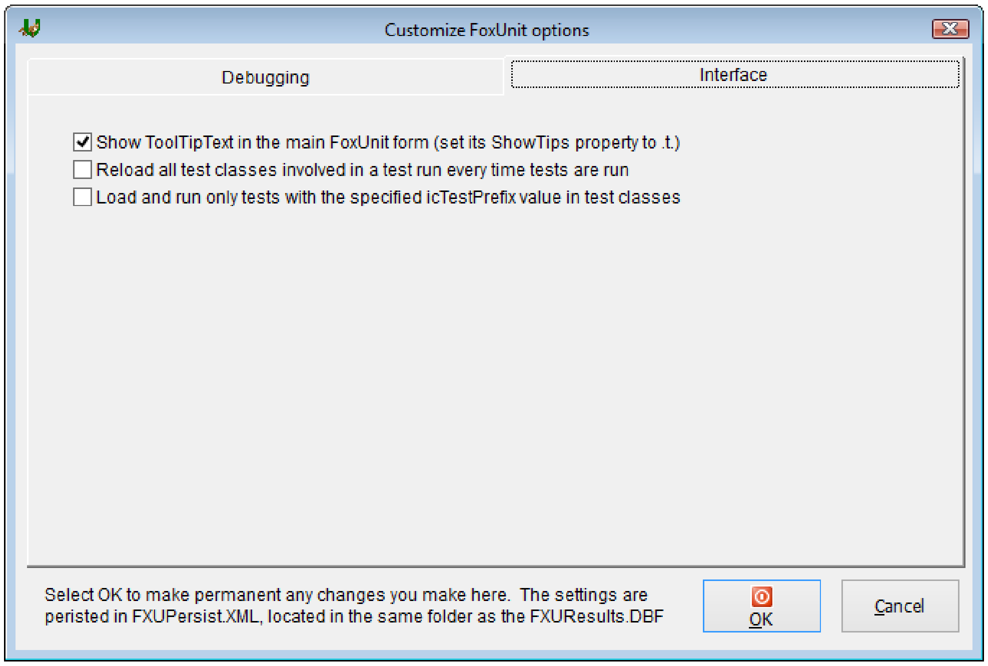
```
THIS.Setup()
TRY
 THIS.RunTest()
CATCH TO loEx
 THIS.HandleException(...)
ENDTRY
THIS.TearDown()
```

(This is the behavior you get if you select the above checkbox)

As shown on the right, if you select the checkbox above, there is no TRY..CATCH exception handling for the Setup() and TearDown() code.
Errors in Setup() and TearDown() methods invoked the native VFP Cancel/Ignore/Suspend dialog.

Select OK to make permanent any changes you make here. The settings are peristed in FXUPersist.XML, located in the same folder as the FXUResults.DBF

Figure 8. The Debugging page of the Options dialog specifies error handling and debugging options for test runs.



☑ Show ToolTipText in the main FoxUnit form (set its ShowTips property to .t.)
☐ Reload all test classes involved in a test run every time tests are run
☐ Load and run only tests with the specified icTestPrefix value in test classes

Select OK to make permanent any changes you make here. The settings are peristed in FXUPersist.XML, located in the same folder as the FXUResults.DBF

Figure 9. The Interface page of the Options dialog controls some UI settings.

Assertions

FoxUnit supports the following assertions:

- AssertFalse: asserts that the expression passed as the first parameter evaluates to false.

- AssertTrue: asserts that the expression passed as the first parameter evaluates to true.
- AssertEquals: asserts that the expressions passed as the first two parameters evaluate to the same value. Pass an optional message (see below) as the third parameter and .T. as the fourth parameter to do a case-insensitive string comparison.
- AssertNotEmpty: asserts that the expression passed as the first parameter evaluates to a non-empty value.
- AssertNotNull: asserts that the expression passed as the first parameter evaluates to a non-null value.
- AssertNotNullOrEmpty: asserts that the expression passed as the first parameter evaluates to a non-null and non-empty value.

All of these methods accept an additional parameter: the message to display in the Failures and Errors tab when the assertion fails.

Mock objects

One thing that complicates testing is making sure you're only testing the method you think you're testing. For example, consider the updated LogFileProcessor class shown in **Listing 6**, taken from NewLogFileProcessor.PRG.

Listing 6. This updated LogFileProcessor uses a helper object.

```
define class LogFileProcessor as Custom
   cErrorMessage = ''
   oParser = .NULL.
   nLogType = 1

   function IsValidFile(tcFileName)
      local llReturn
      do case
         case vartype(tcFileName) <> 'C' or empty(tcFileName)
            This.cErrorMessage = 'Invalid filename'
            llReturn = .F.
         case lower(justext(tcFileName)) <> 'log'
            This.cErrorMessage = 'Not a log file'
            llReturn = .F.
         case not file(tcFileName)
            This.cErrorMessage = 'File does not exist'
            llReturn = .F.
         case empty(filetostr(tcFileName))
            This.cErrorMessage = 'File is empty'
            llReturn = .F.
         otherwise
            llReturn = .T.
      endcase
      return llReturn
   endfunc
   function ImportLog(tcFileName)
      if This.nLogType = 1
         This.oParser = createobject('XMLFileParser')
      else
         This.oParser = createobject('CSVFileParser')
      endif
      lnSelect = select()
      lcCursor = sys(2015)
      lnResult = -1
      llOK     = This.oParser.ParseFile(tcFileName, lcCursor)
      if llOK
         lnResult = This.ProcessFile(lcCursor)
         use in (lcCursor)
      else
         This.cErrorMessage = This.oParser.cErrorMessage
      endif
      select (lnSelect)
      return lnResult
   endfunc

   function ProcessFile(tcCursor)
      return reccount(tcCursor)
   endfunc
enddefine

define class FileParser as Custom
   cErrorMessage = ''

   function ParseFile(tcFile, tcCursor)
      return ''
   endfunc
enddefine

define class CSVFileParser as FileParser
   function ParseFile(tcFile, tcCursor)
      create cursor (tcCursor) (DATE D, TYPE N(1), TEXT C(80))
      append from (tcFile) delimited
      return .T.
   endfunc
enddefine

define class XMLFileParser as FileParser
   function ParseFile(tcFile, tcCursor)
      try
         xmltocursor(tcFile, tcCursor)
         llReturn = .T.
      catch to loException
         This.cErrorMessage = loException.Message
         llReturn = .F.
      endtry
      return llReturn
   endfunc
enddefine
```

The ImportLog method uses a helper object, an instance of either CSVFileParser or XMLFileParser, depending on the type of log file, to convert the text log file into a cursor it can process. **Listing 7** shows a unit test that tests whether ImportLog returns the expected value for a sample XML log.

Listing 7. Test that ImportLog returns the expected value.

```
function TestImportLog_ImportXML_ReturnsCorrectValue
   text to lcLog noshow pretext 2
      <logs>
         <log>
            <date>01/01/2013</date>
            <type>2</type>
            <text>This is a log entry</text>
         </log>
         <log>
            <date>01/02/2013</date>
            <type>1</type>
            <text>This is another log entry</text>
         </log>
      </logs>
   endtext
   strtofile(lcLog, 'x.log')
   lnActual   = This.oObjectToTest.ImportLog('x.log')
   lnExpected = 2
   This.AssertEquals(lnExpected, lnActual, This.oObjectToTest.cErrorMessage)
endfunc
```

This test works as expected. However, it isn't really a good test because it's testing more than the functionality of the ImportLog method: it's also inadvertently testing the functionality of the XMLFileParser. In other words, we're doing an integration test, which Roy Osherove defines as testing two or more dependent software modules as a group. While it has its place, the problem with integration testing is that there are many points of failure so it's hard to find the source of the problem if a test fails. Here, our goal is to just test the LogFileProcessor class, not other classes it happens to interact with.

For example, suppose the XML used in the test is flawed. In that case, the XMLTOCURSOR() call in XMLFileParser.ParseFile would throw an exception. But that's not really the concern of ImportLog. All we want to know when we test it is that it did its job.

In order to test just ImportLog, we have to take the behavior of the helper object out of the equation. The way we do that is by using a mock object instead.

There are actually several types of mock objects—stubs, mocks, and fakes, to name a few—but the important feature they all share is that they take the place of a real object, providing the same interface (not user interface but properties and methods) of the replaced object but having no or very little behavior. I won't distinguish between the types and use "mock" generically.

Before we look at a mock object, we should discuss a design concept called dependency injection that isn't directly part of testing but impacts the testability of your code.

Dependency injection

Notice in Listing 6 that ImportLog instantiates either XMLFileParser or CSVFileParser into oParser. That causes a problem for testing: we can't use a mock for the helper object because the method we want to test only uses the hard-coded classes. (This design also limits the capabilities of ImportLog. What if we want to handle some other type of log file that needs a different parsing object? We'd have to change the IF statement in ImportLog to a CASE and add progressively more cases as we add new log types.)

Wikipedia defines dependency injection as "a software design pattern that allows the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time." This is generally implemented by having the client or caller of a class instantiate dependencies on behalf of the class. There are several ways to do this:

- Set a property to the dependency. Rather than having ImportLog instantiate some class into oParser, the caller is responsible for doing that; ImportLog simply expects that it's been done.
- Pass the dependency as a parameter to the Init method of the class. The Init method of NewLogFileProcessor could look something like this:

  ```
  function Init(toParser)
     This.oParser = toParser
  endfunc
  ```

- Pass the dependency as a parameter to the method using it. This isn't used as often as the first two choices because if many methods use the dependency, it must be passed to each of them.
- Have properties of the class containing the name and library to instantiate the dependency from. I use this technique a lot because it's very flexible: change the values of the properties and a different class gets instantiated. The only issue with this is if the dependency has to be instantiated in Init, you can't change the properties at run-time, only design-time.
- Use a class factory. This is often used in data-driven frameworks like Visual FoxExpress. The class calls a method of the class factory, telling it that it needs a log file parser. The class factory decides what class to instantiate, often by looking it up in a table the developer (or even end-user) can configure, and returns the object, which the class then uses. While this is a popular mechanism in VFP applications, it makes testing more difficult because the test has to configure the class factory properly and it ultimately tests the functionality of the class factory in addition to the class under test.

Let's use the first mechanism. Here's the change in ImportLog to support that:

```
if vartype(This.oParser) <> 'O'
```

```
   This.cErrorMessage = 'oParser was not set to a parser object'
   return -1
endif
*!* if This.nLogType = 1
*!*    This.oParser = createobject('XMLFileParser')
*!* else
*!*    This.oParser = createobject('CSVFileParser')
*!* endif
```

The only change needed in the test is to instantiate the desired parser object. We could use:

```
This.oObjectToTest.oParser = newobject('XMLFileParser', 'NewLogFileProcessor.prg')
```

but that still uses the XMLFileParser and doesn't resolve the problem we're trying to solve using dependency injection. Instead, let's create a mock file parser class.

Creating a mock

The purpose of a mock object is to have the interface of a dependency object without having its behavior. Because it has no or little behavior, it allows interaction between the objects without the overhead of additional complications.

For example, look at what the ParseFile methods of CSVFileParser and XMLFileParser do: they accept a file name and a cursor name as parameters, create a cursor with the name of the second parameter, and return .T. or .F. What they do internally differs, but from LogFileProcessor's viewpoint, that's immaterial. So, let's create a mock class that does the minimum necessary to work with LogFileProcessor. See **Listing 8**.

Listing 8. MockFileParser is a mock class to use with LogFileProcessor.

```
define class MockFileParser as Custom
   function ParseFile(tcFile, tcCursor)
      create cursor (tcCursor) (DATE D, TYPE N(1), TEXT C(80))
      append blank
      return .T.
   endfunc
enddefine
```

Listing 9 shows the updated test. Notice that this test is simpler than **Listing 7**: it isn't concerned with having valid XML content or what that content consists of. Since MockFileParser doesn't care about files or their contents but simply creates a dummy cursor, we're freed from the issue of testing the parser's behavior at the same time as LogFileProcessor's.

Listing 9. The updated test using the mock object is simpler.

```
function TestImportLog_Import_ReturnsCorrectValue_UsingMock
   This.oObjectToTest.oParser = createobject('MockFileParser')
   lnActual   = This.oObjectToTest.ImportLog('x.log')
   lnExpected = 1
   This.AssertEquals(lnExpected, lnActual, This.oObjectToTest.cErrorMessage)
endfunc
```

Automating mock creation with FoxMock

The mock object in **Listing 8** was easy to create but if you have a lot of dependencies or the dependency objects are complex, it can be a lot more work to create mocks for all of them.

Other development environments such as .Net and Java have frameworks that create mock objects automatically. Fortunately for us VFP developers, Christof Wollenhaupt created a mocking framework named FoxMock. According to Christof, the purpose of FoxMock is to "dynamically create objects that simulate existing objects but do not duplicate their behavior." He presented FoxMock at Southwest Fox 2012; if you don't have access to the downloads for his session, you can download FoxMock from https://bitbucket.org/cwollenhaupt/foxpert.tools.foxmock/src.

Let's look at a little more complicated class to test: a customer business object class shown in **Listing 10**, taken from BizObject.PRG. The GetCustomer method uses a database connection object to retrieve data from a database. Different type of connection objects could deal with VFP, SQL Server, or other types of databases, so the business object doesn't have to worry about that.

Listing 10. This customer business class collaborates with a database connection object.

```
define class CustomerBO as Custom
   oConnection = .NULL.
   cErrorMessage = ''

   function Init(toConnection)
      This.oConnection = toConnection
   endfunc

   function GetCustomer(tcID)
      loCustomer = .NULL.
      lnSelect   = select()
      if This.oConnection.Connect()
         private lcID
         lcID = tcID
         if This.oConnection.Execute('select * from Customers where ' + ;
            'CustomerID=?lcID')
            if reccount() > 0
               scatter name loCustomer
            else
               This.cErrorMessage = 'Record not found'
            endif
            use
         else
            This.cErrorMessage = 'SQL statement failed: ' + ;
               This.oConnection.cErrorMessage
         endif
      else
         This.cErrorMessage = 'Could not connect to database: ' + ;
            This.oConnection.cErrorMessage
      endif
      select (lnSelect)
      return loCustomer
   endfunc
enddefine
```

To use this class, we need a connection class. However, it may not exist yet or it may connect to a database we haven't created yet. But for testing, we can use a mock object so the connection class doesn't have to exist.

Listing 11 shows a FoxUnit test class for CustomerBO. The Setup method instantiates FoxMock into the oMock property so it's available to all tests. Each test in this class uses FoxMock to create a mock connection object that satisfies the requirements of CustomerBO.

Listing 11. The tests for CustomerBO use FoxMock to automatically create mock connection objects.

```
define class bizobjecttests as FxuTestCase of FxuTestCase.prg
   #if .f.
      local this as bizobjecttests of bizobjecttests.PRG
   #endif

   icTestPrefix = 'Test'
   oObjectToTest = .NULL.
   oMock = .NULL.

   function Setup
      This.oMock = newobject('FoxMock', 'FoxMock.prg')
      This.oObjectToTest = newobject('CustomerBO', 'BizObject.prg')
   endfunc

   function TearDown
   endfunc

   function TestGetCustomer_CantConnect_ReturnsNull
      loConn = This.oMock.New.Property('cErrorMessage').Is( ;
         "'cannot connect'").Method('Connect').Returns('.F.')
      This.oObjectToTest.oConnection = loConn
      loActual = This.oObjectToTest.GetCustomer()
      This.AssertTrue(isnull(loActual))
   endfunc

   function TestGetCustomer_CantExecute_ReturnsNull
      loConn = This.oMock.New.Property('cErrorMessage').Is( ;
         "'cannot execute'").Method('Connect').Returns('.T.').Method( ;
         'Execute').Returns('.F.')
      This.oObjectToTest.oConnection = loConn
      loActual = This.oObjectToTest.GetCustomer()
      This.AssertTrue(isnull(loActual))
   endfunc

   function TestGetCustomer_NoRecords_ReturnsNull
      create cursor Temp (Field1 C(1))
      loConn = This.oMock.New.Method('Connect').Returns('.T.').Method( ;
         'Execute').Returns('.T.')
      This.oObjectToTest.oConnection = loConn
      loActual = This.oObjectToTest.GetCustomer()
      This.AssertTrue(isnull(loActual))
      use in select('Temp')
   endfunc

   function TestGetCustomer_FindRecord_ReturnsObject
      create cursor Temp (Field1 C(1))
      insert into Temp values ('x')
      loConn = This.oMock.New.Method('Connect').Returns('.T.').Method( ;
         'Execute').Returns('.T.')
      This.oObjectToTest.oConnection = loConn
      loActual = This.oObjectToTest.GetCustomer()
      This.AssertNotNull(loActual)
      use in select('Temp')
   endfunc

   function TestGetCustomer_FindRecord_ReturnsValue
      create cursor Temp (Field1 C(1))
      insert into Temp values ('x')
      loConn = This.oMock.New.Method('Connect').Returns('.T.').Method( ;
         'Execute').Returns('.T.')
      This.oObjectToTest.oConnection = loConn
      loActual = This.oObjectToTest.GetCustomer()
      This.AssertEquals('x', loActual.Field1)
      use in select('Temp')
   endfunc
enddefine
```

Let's look at the first test. TestGetCustomer_CantConnect_ReturnsNull. As its name suggests, it tests that GetCustomer returns null if the connection object can't connect to the database. So, we create a mock connection object whose Connect method returns .F. (it also has a cErrorMessage property that contains "cannot connect" because GetCustomer uses that property).

```
loConn = This.oMock.New.Property('cErrorMessage').Is( ;
   "'cannot connect'").Method('Connect').Returns('.F.')
```

Let's break this down. This.oMock.New creates a new object. Property('cErrorMessage') adds a cErrorMessage property to that object and Is("'cannot connect'") specifies the value of the property (note that since the specified value will be evaluated, it must be in quotes, so a string must have a double set of quotes). Method('Connect') adds a Connect method to the object and Returns('.F.') specifies that the method returns .F. Now we simply tell the CustomerBO object being tested to use the mock object:

```
This.oObjectToTest.oConnection = loConn
```

Since the connection object's Connect method returns .F., calling GetCustomer results in a null value.

TestGetCustomer_FindRecord_ReturnsObject is a little more complicated. We want both the Connect and Execute methods of the mock connection object to return .T. and don't need the cErrorMessage property:

```
loConn = This.oMock.New.Method('Connect').Returns('.T.').Method( ;
   'Execute').Returns('.T.')
```

However, since the mock object has no other behavior, the test has to create and populate the cursor used by GetCustomer. As with other tests, we then set the oConnection property of the object being tested to the mock object, call GetCustomer, and test that an object was returned.

As you can see, configuring FoxMock to create an object with the desired interface is easy. FoxMock actually has a lot more capability than I show here, so I recommend reading Christof's white paper to discover what else you can do with it.

Test-driven development

Test-driven development, or TDD, is a methodology of software development in which tests are written along with, and sometime before, the code being tested. The idea of TDD is that you write a test that fails because the code it's testing doesn't exist yet or does nothing, then change the code so the test passes, write more tests that fail, write more code so tests pass, and so on. The tests are written to test the functionality of the spec, not the code, and the code is written so tests pass. Thus, the code implements the functionality of the spec because the tests say it does.

You don't have to use TDD to take advantage of unit testing. I personally don't (currently anyway) follow TDD but I do tend to write tests very soon after I've started working on a method.

Testing UI

Testing your UI is more complicated than testing "engine" code because it relies on user interaction such as typing in a textbox or clicking a button. Although VFP has a MOUSE command to simulate mouse movement and clicks and KEYBOARD to simulate typing, using them can result in fragile tests; any changes to the locations of control causes the tests to break even if the behavior hasn't changed. Fortunately, because of how VFP implements event handlers, you can simulate entering text by writing to the Value property of controls or clicks by calling the Click method. Still, testing UI is a lot more work than testing non-UI code, so that's another reason to minimize the amount of code in the UI and instead have UI events call methods of other objects.

Testing legacy code

It can be difficult to write unit tests for legacy code for several reasons:

- The code usually doesn't have "seams," places where mock objects can replace real ones.

- Refactoring code that works only so tests can be created seems like a waste of time.
- Refactoring code that has no tests to begin with is a risky business.
- It's difficult to determine where to begin.
- Clients usually don't want to pay for something that doesn't add new features to the application.

Roy Osherove suggests starting with integration tests before refactoring so you can test whether the refactoring broke the functionality or not, and then add unit tests in either an "easy first" or "hard first" manner, depending on your team's experience with unit testing principles.

Guidelines for writing good unit tests

Here are some guidelines I use for writing good unit tests. I've taken these guidelines from Roy Osherove's book, various blogs, and my own experiences.

- Keep tests simple. A unit test should test only one aspect of one method of one class.
- Related to the previous point, ideally a test should have a single assertion. Usually more than one assertion is a sign that the test is testing more than one thing. I'll admit this is a guideline I occasionally break (five points to Gryffindor if you find where I've done that in this document) but I usually don't like the "smell" of the test when I do. If you do break this rule, be sure to use the optional message parameter of the assert methods so you know exactly which one failed.
- Use descriptive names for your tests. Help the developer coming after you (or even yourself, looking at the test code months later) by naming exactly what each test is supposed to test for and what the result of the test should be.
- Avoid logic in tests. If a test has conditional code (IF or CASE statements) or loops (DO WHILE, SCAN, or FOR), the code risks being complex and introducing its own set of bugs. Now you have a second code base to maintain!
- Only test for what the code is supposed to do, not what it does. It's better to write a unit test by looking at the specs for the code rather than reading the actual code. The problem with writing tests based on what the code does is that you're testing implementation rather than behavior. What if the internals of the code change but the interface and behavior don't? Now tests testing internal implementation are broken even though the behavior hasn't changed. If method A calls private method B, which has its own behavior, it might be better to make method B public so you can write unit tests just for method B.
- Use helper methods to eliminate duplication. Duplicate code in tests is as bad as it is in application code. Refactor your tests as necessary to eliminate duplicate code. We saw earlier that I created helper methods to instantiate the object under test. Other helper methods might set up commonly used mock objects. Don't forget that the Setup and Teardown methods are really built-in helper methods available for your use.

- To expand upon something in the previous point, you must ensure all tests run in isolation. Suppose you set up certain external conditions, such as creating a log file in our examples, in one test and don't reset them when the test is done. That can affect other tests in confusing (to the developer) ways. The worst case is if the tests give different results depending on whether they're run alone or along with other tests because now you're not debugging your application code, you're debugging your tests! If your objects have certain external expectations, make sure every test has a clean starting point, not one that could be polluted from a previously run test.

  For example, the second test fails because it assumes x.log doesn't exist, but the first test created and didn't delete it:

  ```
  function TestIsValidFile_ValidFile_ReturnsTrue
     strtofile('some text', 'x.log')
     llResult = This.oObjectToTest.IsValidFile('x.log')
     This.AssertTrue(llResult)
  endfunc

  function TestIsValidFile_NonExistentFile_ReturnsFalse
     llResult = This.oObjectToTest.IsValidFile('x.log')
     This.AssertFalse(llResult)
  endfunc
  ```

- Don't tolerate failing tests. If a test fails, there's a reason for it: either the code being tested is broken or the test itself if broken. Find out which it is and fix it. Otherwise, you can't be confident the code you're delivering to your customer completely works or not.

References

Here are some reference materials I used to learn about unit testing:

The Art of Unit Testing, by Roy Osherove, ISBN 978-1-933988-27-6. Although his example are all in C# and he discusses .Net tools, the principles and techniques he describes are universal.

Using FoxUnit for Test-Driven Development, by Andrew MacNeill, http://www.aksel.com/whitepapers/foxunit.htm. This was one of the first white papers written about unit testing in the VFP world and still stands up well today.

Summary

Unit testing is a very important technique in software development. It's a shame that it isn't more prevalent in the VFP world, so hopefully this document will get you started in unit testing so you can share the message with other VFP developers.

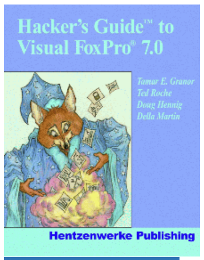
Biography

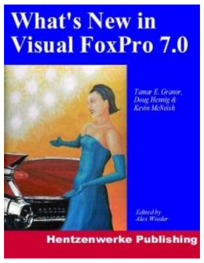
Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the

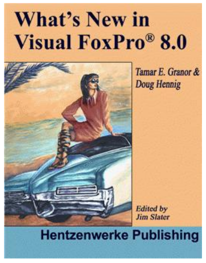
MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of *The Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox and Southwest Xbase++ conferences (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).






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