FoxUnit in Depth

Eric Selje
Salty Dog Solutions, LLC
www.SaltyDogLLC.com
Madison, WI USA
Voice:608-213-9567
Twitter: @EricSelje
Email:Eric@SaltyDogLLC.com

You've got years and years of legacy code to maintain. How can you be sure you're making changes without breaking everything and putting your company and your reputation at risk?

In this session we'll review what Unit Testing is, why you should be doing it, and how to use the VFPX tool FoxUnit to perform unit tests on your code. If it's been a while since you've looked at FoxUnit, you may be pleasantly surprised at how many new features have been added in the last few years, thanks to multiple contributors in the Fox family.

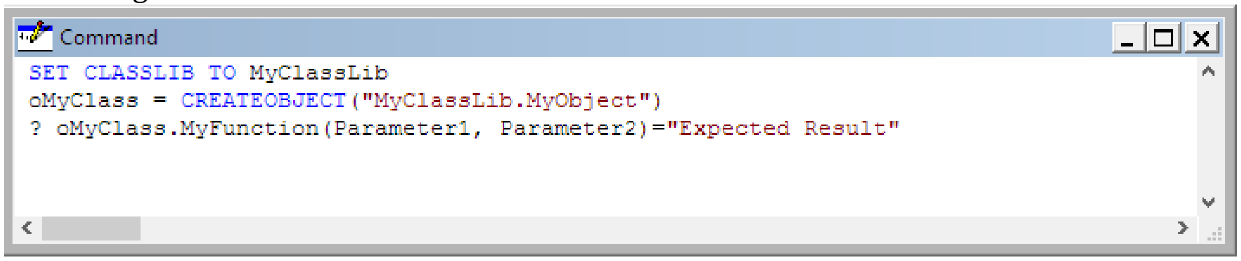
Introduction

Have you ever written a function and just *knew* that it was perfect? It sprung from your fingertips perfectly formed, did exactly everything you needed the first time, and you never needed to touch that code again? If so, unit testing is not for you, and you may stop reading. The rest of us will find unit testing incredibly beneficial.

What is "Unit Testing"

Unit testing is writing code that "tests" your real code. It ensures your code does what you expect, doesn't throw (unexpected) exceptions, and returns the value(s) you want and never the values you don't want.

I used to find myself writing code in FoxPro, then immediately dropping to the Command Window to verify that the code I just wrote worked. Often that would end up looking something like this:



```
SET CLASSLIB TO MyClassLib
oMyClass = CREATEOBJECT("MyClassLib.MyObject")
? oMyClass.MyFunction(Parameter1, Parameter2)="Expected Result"
```

That's unit testing! It's not very *good* unit testing, but it is unit testing. The problem with this code is:

- It's designed to succeed. We were expecting a certain result when we sent in two parameters of a specific type, and we got them so we called it a success. But what happens if you send only one parameter? Or three? Or parameters of unexpected types? Or *nulls*? We didn't test for any of that so we have no clue whether or function is actually robust at all.

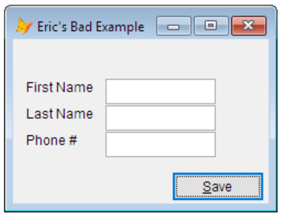
- It's ephemeral. That test code will disappear into the history of the Command Window before the day's over. We could re-run the test by trying to find that code again, but that inefficient. If you're clever you might put it into a little PRG file so you can find it quicker and run it, but that doesn't scale very well.

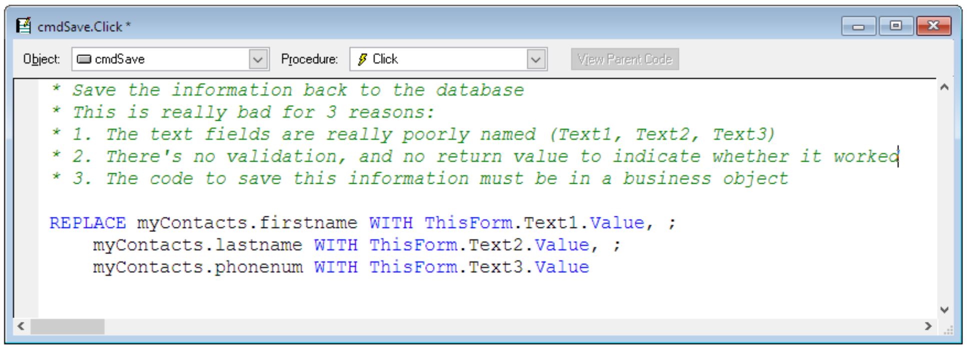
- It's manual. Wouldn't it be much better if you could *automatically* run the unit tests you've written whenever you make a change to your code?

Ok, if that's not a good unit test, let's make it better. A good rule of thumb is that all of the tests for a *specific object* should be in their own isolated test class.

Hold up… what if your code isn't in an object? How can you test, say, the code that's in the Click() event of a button of a form? Simply put, you can't. It's "untestable". Well you *could* but you'd have to create some very awkward code and do other acrobatics to make it work.

That's why we've been told for years that there should *never* be business logic stored in events except a call to a method of a business object. For example, if there's a Save button on the form, and the code to write the data from the form to the table is in the event method of that Save button.





```
* Save the information back to the database
* This is really bad for 3 reasons:
* 1. The text fields are really poorly named (Text1, Text2, Text3)
* 2. There's no validation, and no return value to indicate whether it worked
* 3. The code to save this information must be in a business object

REPLACE myContacts.firstname WITH ThisForm.Text1.Value, ;
    myContacts.lastname WITH ThisForm.Text2.Value, ;
    myContacts.phonenum WITH ThisForm.Text3.Value
```

And if you think moving this code to the form level and then calling ThisForm.Save() is a good idea, I'd say it's a step in the right direction but it really doesn't solve the problem. If you did that you'd be able to call the Save() code from multiple places on the form, e.g. when the form's Close event fires, without resorting to something like

```
ThisForm.cmdSave.Click()  && Bad Bad Bad!!
```

But it still leaves the code to save the data in the form itself, which is still considered untestable because you'd have to instantiate the form, a user-interface object, in order to test the method. Testing the user-interface is *not* unit testing.

Instead, create a class that mediates your user interface and your database, known as a "business object," and put the Save code in there. If you haven't used business objects before check out Rick Strahl's wwBusiness[i] and Tamar Granor's whitepaper from Southwest Fox 2009[ii].

Writing testable code is key to good software engineering. It might seem encumbering to have your object's event code call a method of the form which invokes a business object, but in the end it creates much cleaner separation of concerns and testable code.

Let's get back to how we write good unit tests. For these examples in this whitepaper I'm going to use the **FoxTypes** class I created *(see Appendix A, and also in the session downloads and on my github page, https://github.com/eselje/FoxTypes)* that emulates C#'s String.Format() function in Visual FoxPro. It's instantiated like this:

```
SET PROCEDURE TO STRING.prg ADDITIVE
* This makes the STRING object available everywhere, like C#
ADDPROPERTY(_vfp, "STRING", CREATEOBJECT("String”))
```

and used like this

```
? _VFP.STRING.Format("Welcome to Southwest Fox {0}!", 2016)
```

Welcome to Southwest Fox 2016!

Pretty cool eh? You can also add up to 9 parameters too and even mix up the types and order of the parameters. E.g.

```
? _VFP.STRING.Format("Precons for {3} {0} are on {1:D} and cost a mere {2:C}. ", ;
   2016, {^2016/09/22}, 99, "Southwest Fox/xBase++")
```

Precons for Southwest Fox/xBase++ 2016 are on Sep 22, 2016 and cost a mere $99.

How would we write unit tests for this String class? Let's start by taking a look at what constitutes a good unit test:

1. A good unit test should test one thing and one thing only, and the result of the test should be pass or fail.

2. A good unit test does not rely on other unit tests. Each test should be self-contained, not have dependencies or leave remnants of itself, should be runnable in any order, and not have side effects. To that end, if your function relies on something like a call to a REST function or a database, we must 'mock' that call in the unit test. Why?

   In the instance of a call to a REST function, your unit test's job is not to test that REST function. In the instance of the database call, that expects that there's a database available ("dependencies"), and if you alter the database that will violate the "no side effects" rule.

3. A good unit test should be easy to read and quick to run.

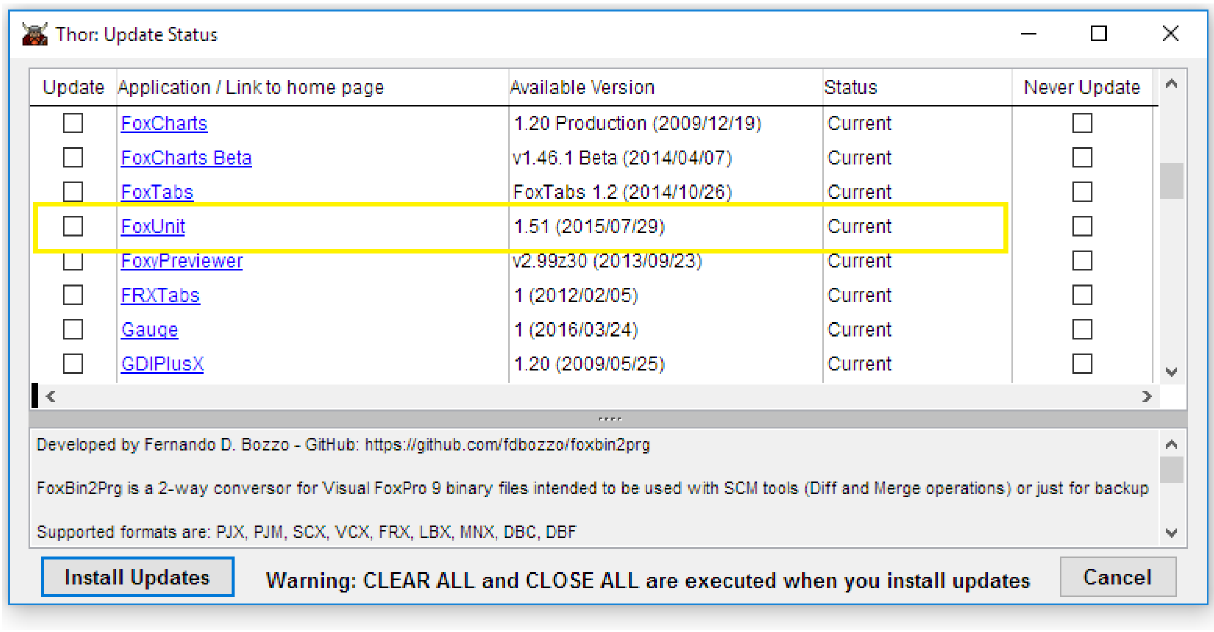
4. Unit tests should be well organized, with one "test class" for each object you're testing. There can be *multiple* tests for any of the methods in your test class and in fact that's encouraged! You want to find out what happens when you don't pass the right number of parameters, or pass parameters of the wrong type, or *nulls*. Can your function handle extra large numbers or negative numbers? Throwing these *edge cases* at your function is encouraged when you're unit testing.

Let's take a look at how FoxUnit helps us accomplish these goals.

Installing FoxUnit

FoxUnit is the only (as far as I'm aware) unit testing *framework* for Visual FoxPro. It was initially written by VisionPace Corp. but has been submitted to VFPx, the open-source repository for Visual FoxPro projects.

There are a couple of ways to get the FoxUnit bits onto your system. If you're using Thor (and you really should be using Thor), FoxUnit is one of the many applications you can install.[iii]



Update	Application / Link to home page	Available Version	Status	Never Update
☐	FoxCharts	1.20 Production (2009/12/19)	Current	☐
☐	FoxCharts Beta	v1.46.1 Beta (2014/04/07)	Current	☐
☐	FoxTabs	FoxTabs 1.2 (2014/10/26)	Current	☐
☐	FoxUnit	1.51 (2015/07/29)	Current	☐
☐	FoxyPreviewer	v2.99z30 (2013/09/23)	Current	☐
☐	FRXTabs	1 (2012/02/05)	Current	☐
☐	Gauge	1 (2016/03/24)	Current	☐
☐	GDIPlusX	1.20 (2009/05/25)	Current	☐

Developed by Fernando D. Bozzo - GitHub: https://github.com/fdbozzo/foxbin2prg

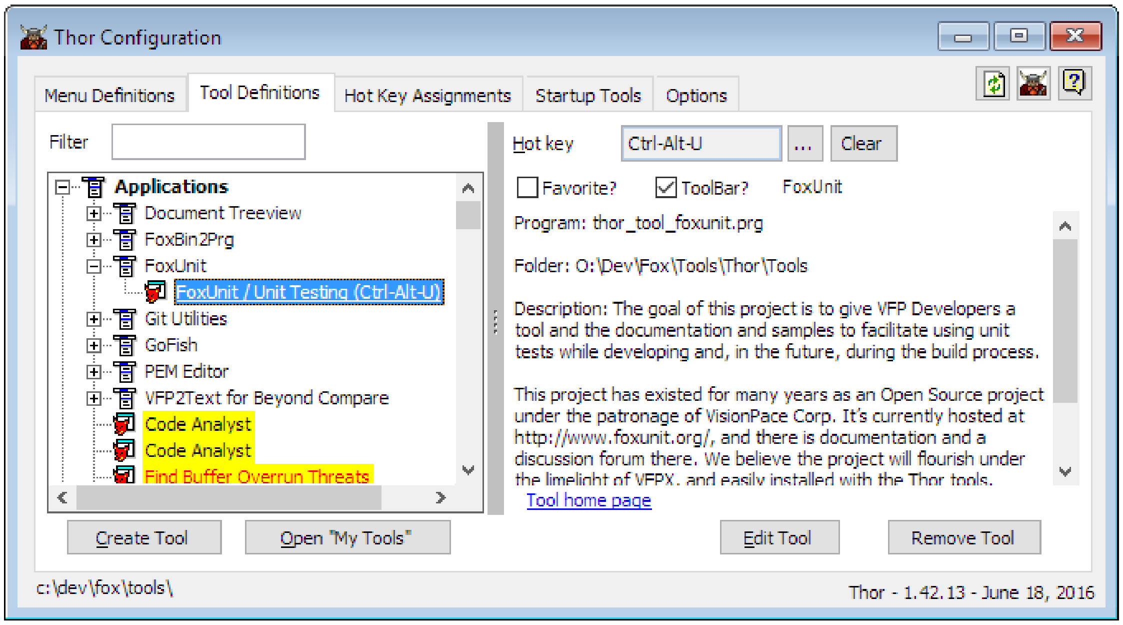
FoxBin2Prg is a 2-way conversor for Visual FoxPro 9 binary files intended to be used with SCM tools (Diff and Merge operations) or just for backup

Supported formats are: PJX, PJM, SCX, VCX, FRX, LBX, MNX, DBC, DBF

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

Another tremendous advantage of using Thor is that any of its applications can be added to your toolbar as well as "hotkeyed" through the Configuration screen, so starting FoxUnit is always as easy as Ctrl+Alt+U (in my case).

I'd



Hot key: Ctrl-Alt-U

☐ Favorite? ☑ ToolBar? FoxUnit

Program: thor_tool_foxunit.prg

Folder: O:\Dev\Fox\Tools\Thor\Tools

Description: The goal of this project is to give VFP Developers a tool and the documentation and samples to facilitate using unit tests while developing and, in the future, during the build process.

This project has existed for many years as an Open Source project under the patronage of VisionPace Corp. It's currently hosted at http://www.foxunit.org/, and there is documentation and a discussion forum there. We believe the project will flourish under the limelight of VFPX, and easily installed with the Thor tools.

Tool home page

c:\dev\fox\tools\

Thor - 1.42.13 - June 18, 2016

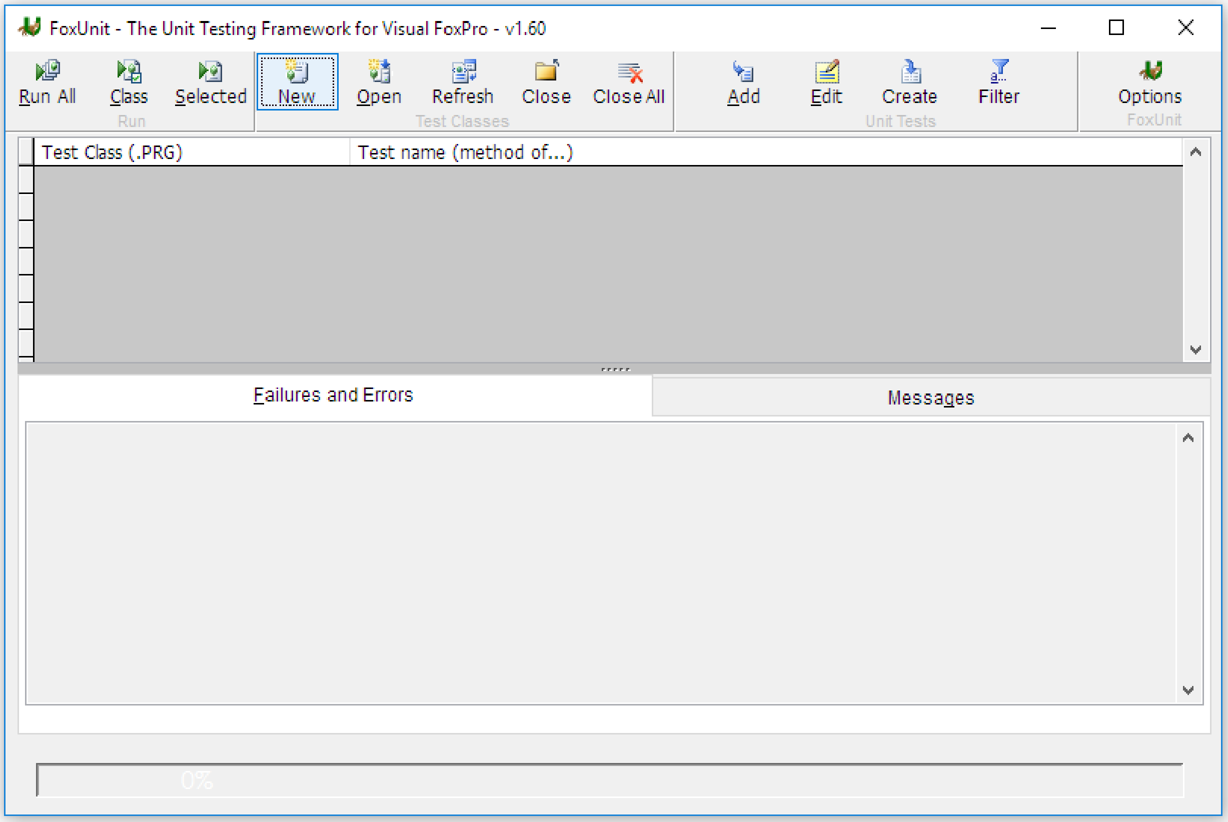
recommend installing FoxUnit via Thor, but if you must you can also get the bits directly from VFPx (http://vfpx.codeplex.com/releases). You can actually get all the source code for FoxUnit on that site as well (more on that later).

If you download FoxUnit from VFPx, you start it by typing

DO <path>\FoxUnit.app

A Tour of FoxUnit

When you first fire it up there's a form showing all of the tests it can find in the *tests*\ subfolder of the current folder (in this example, there are none yet).



Here are the components of this User Interface:

Organize Tests

In FoxUnit, tests are organized into Test *Classes*, where each test class is a PRG on disk, ideally in a Tests\ subfolder below the class library that's being tested. Each test class should contain all of the unit tests for one class. In the FoxTypes class library, for example, I have the String class, so all of my unit tests for the String class should be in one FoxUnit class. If I later decide to create a DateTime class in FoxTypes, the tests for that would go in a separate test class.

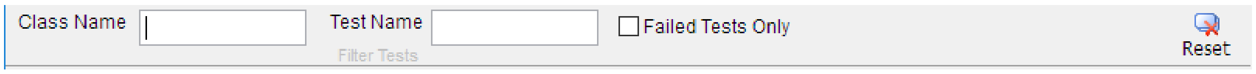
I like to name my test classes something like *StringTests* so I can easily see what class I'm testing. I might even go with FoxTypes_String_Tests to ensure I don't have any confusion at all.

FoxUnit allows you to open multiple Test Classes at once, which is handy if you want to test all of the classes in your class library at the same time. For practical reasons you may want to limit how many Test Classes you load simultaneously, but theoretically the limit is quite large.

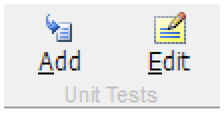
Run Tests

Click on of these buttons to run every test in all of the loaded Test classes, just the tests in the same Test Class as the selected test, or just the one selected test. If your tests are fast it should be no hardship to run All tests every time but as they accumulate you may find it saves you time to run a subset.

If you do have a large number of tests that you've run you can click the "Filter" button to bring up the Filter panel and whittle your test results down Filter by the Class Name, Test Name, or show only the Failed Tests, which are the ones that need your attention.

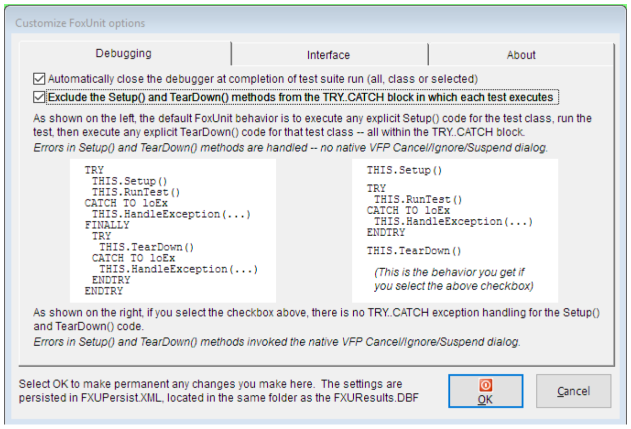


Write Tests



The "Add" button will drop a template of a new test into your test class for you to complete. "Edit" will jump you straight to the selected test in your test class for you to modify (same as Double+Clicking). At this point there's no "Delete" button but if you Edit you can simply delete all of the code in the test method, and when you close the test class the screen will refresh and the deleted test will be removed from the list.

Preferences



Customize FoxUnit options

Debugging | Interface | About

[✓] 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)

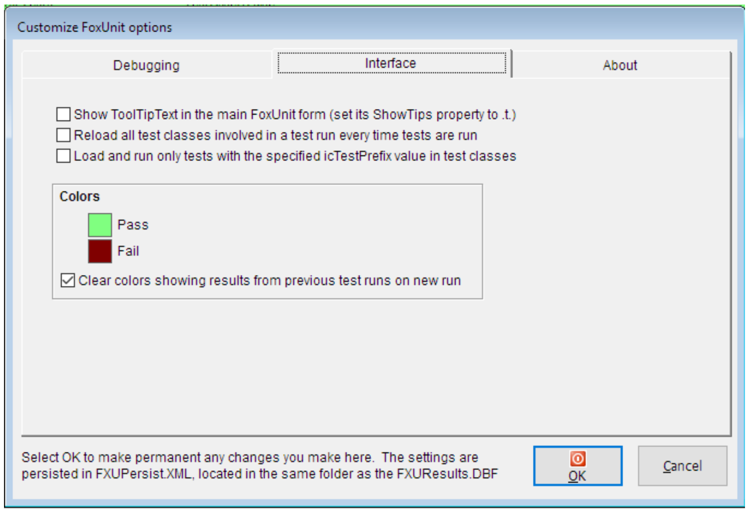
As shown on the right, if you select the checkbox above, there is no TRY..CATCH exception handling for the Setup() and TearDown() code.
Errors in Setup() and TearDown() methods invoked the native VFP Cancel/Ignore/Suspend dialog.

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

FoxUnit gives you many options to customize your preferences. On the *Debugging* tab you can opt whether to automatically close the debugger after you run tests. This is a nice option because the debugger can be handy while you're running tests, but may get in the way afterwards.

You can also choose whether to run the Setup() and Teardown() code outside of the Try…Catch. This will make more sense later when we start writing tests, but you probably don't want FoxUnit to consider errors in those methods as errors in your test itself, so I recommend *excluding* them.

On the *Interface*tab you can tweak the user interface options. A new option in FoxUnit 1.6 is the ability to change which color signify success and failure of tests. This is useful for anyone who has red/green color blindness. We also recently added the ability to clear previous test results when re-running tests, which makes it easier to see the progress of the test run.



☐ 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

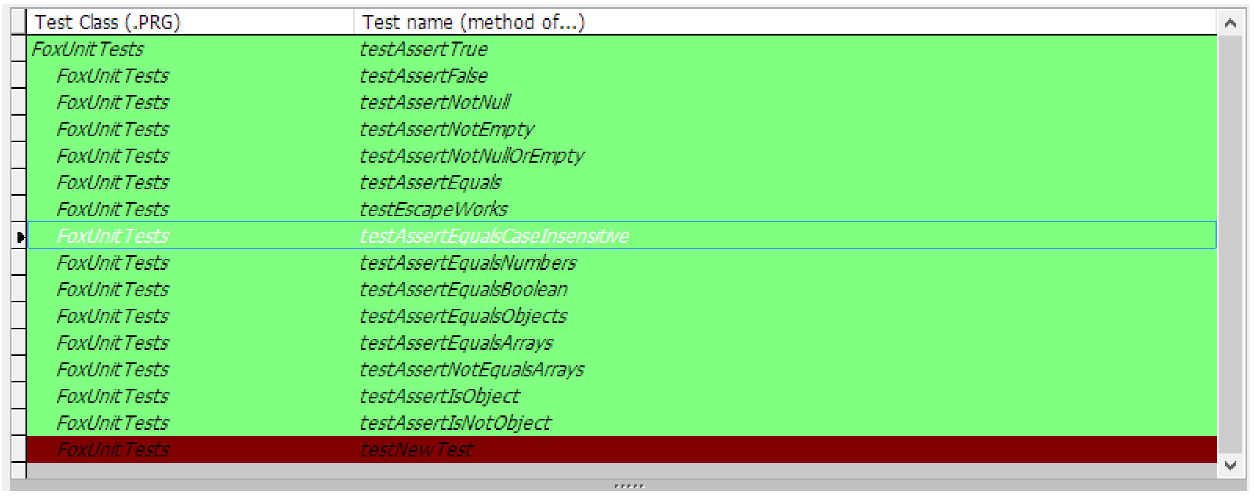
Colors
🟩 Pass
🟥 Fail
☑ Clear colors showing results from previous test runs on new run

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

Besides Setup() and Teardown(), you may have other methods in your test class that aren't actually tests. If you want FoxUnit to ignore those methods, create a naming convention for your tests. For example all of my tests (and this isn't my original idea or anything) start with the word "Test". Then in my test class I can set the *icTestPrefix* value to "Test", check this box on the Interface tab, and FoxUnit will ignore all the methods in my test class that do not start with "Test".

Tests

The middle panel of the FoxUnit form shows the Tests that are loaded.

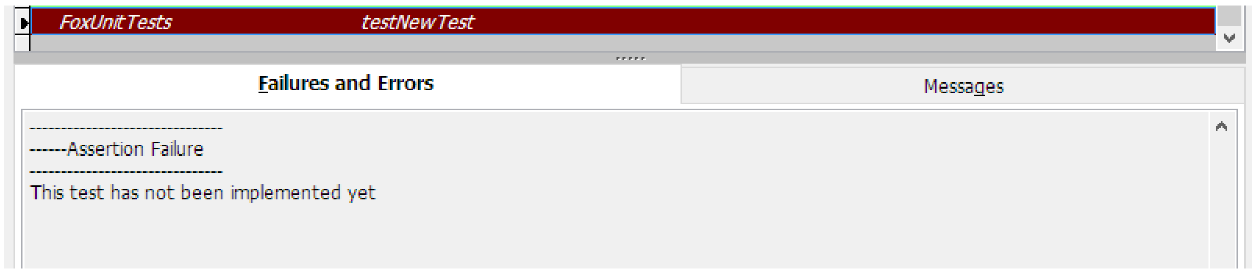


Test Class (.PRG)	Test name (method of...)
FoxUnitTests	testAssertTrue
FoxUnitTests	testAssertFalse
FoxUnitTests	testAssertNotNull
FoxUnitTests	testAssertNotEmpty
FoxUnitTests	testAssertNotNullOrEmpty
FoxUnitTests	testAssertEquals
FoxUnitTests	testEscapeWorks
FoxUnitTests	testAssertEqualsCaseInsensitive
FoxUnitTests	testAssertEqualsNumbers
FoxUnitTests	testAssertEqualsBoolean
FoxUnitTests	testAssertEqualsObjects
FoxUnitTests	testAssertEqualsArrays
FoxUnitTests	testAssertNotEqualsArrays
FoxUnitTests	testAssertIsObject
FoxUnitTests	testAssertIsNotObject
FoxUnitTests	testNewTest

By default green indicates that the test passed the last time it was run, and red indicates it failed. I've customized my colors slightly here but you get the idea. In the example above (which are indeed the tests for FoxUnit itself) all of the tests passed except the last one on the screen.

Errors

If any of the tests have errors, you can see those in the lower panel, along with a message that you create when you write your tests.



FoxUnitTests testNewTest

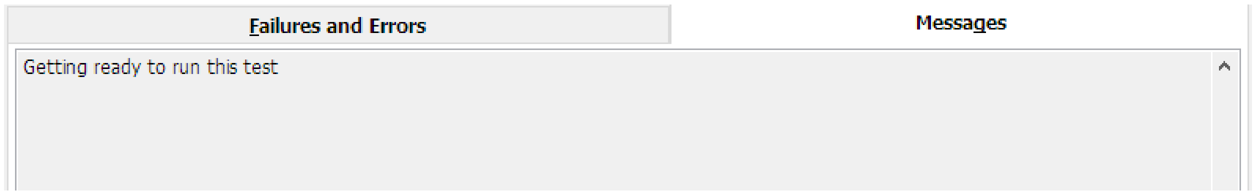
Failures and Errors | Messages

```
--------------------------------
------Assertion Failure
--------------------------------
This test has not been implemented yet
```

In this example there wasn't so much a failure of a test as it just hasn't been implemented yet. Note the splitter bar between the tests and the errors, allowing you to resize the panels to your liking. You can also hit Ctrl+S to toggle the size of the tests to take up the whole screen (and hide the results) or back to the default where the screen is split.

Messages

Much like DEBUGOUT, you can insert code into your tests to send messages to yourself.



Failures and Errors | Messages

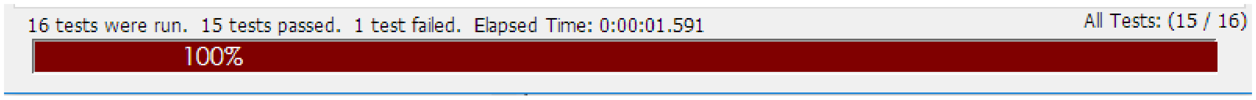
```
Getting ready to run this test
```

Here's the test that created that message:

```
FUNCTION testNewTest
This.MessageOut("Getting ready to run this test")
RETURN This.AssertNotImplemented()
```

Status

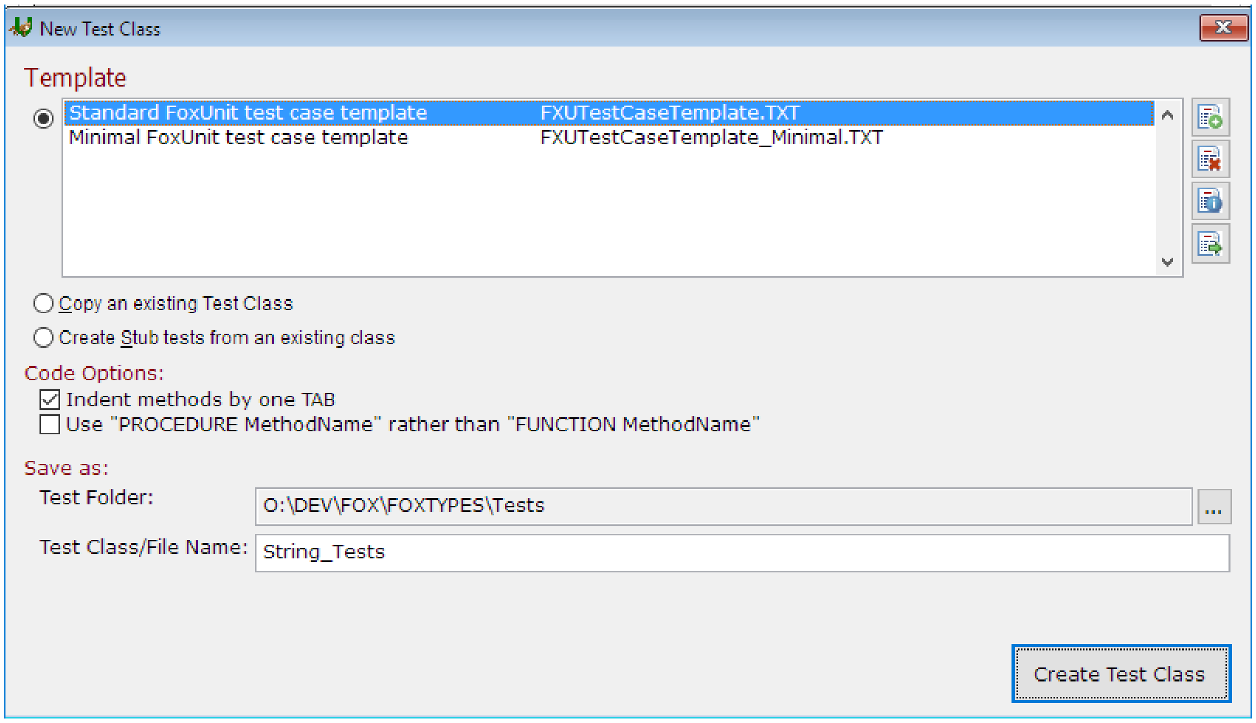
At the very bottom of the screen is the status bar showing the results of the last test run, including elapsed time.



```
16 tests were run.  15 tests passed.  1 test failed.  Elapsed Time: 0:00:01.591     All Tests: (15 / 16)
100%
```

Using FoxUnit

Returning now to the FoxTypes class library, let's begin writing unit tests for our String class to ensure the Format method works as we hope it will. We begin by starting FoxUnit and creating a new Test Class. Clicking the New button brings up the *New Test Class* dialog:


New Test Class

Template

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

(●) [✓ Standard FoxUnit test case template]
( ) Copy an existing Test Class
( ) Create Stub tests from an existing class

Code Options:
[✓] Indent methods by one TAB
[ ] Use "PROCEDURE MethodName" rather than "FUNCTION MethodName"

Save as:
Test Folder: O:\DEV\FOX\FOXTYPES\Tests
Test Class/File Name: String_Tests

Templates

FoxUnit allows you to base your Test Class on your own customized templates, which is very handy if you or your company has them. They might contain some standard unit tests that you want to make sure you include every time. The buttons along the right side of the Templates allows you to create, remove, edit, or add an existing template file.

Templates are merely .TXT files with placeholders for the name of the TestClass. See Appendix B for the standard FoxUnit test case template, which contains many comments, hints for which properties you may want to set and sample test methods. Compare it with Appendix C, the Minimal FoxUnit case template. Until you've become very familiar with the settings I recommend sticking with the standard template.

You can also choose to forego a template and start by copying an existing test class, or creating "stub" tests from a class. I'll describe that new and most excellent feature later.

Code Options

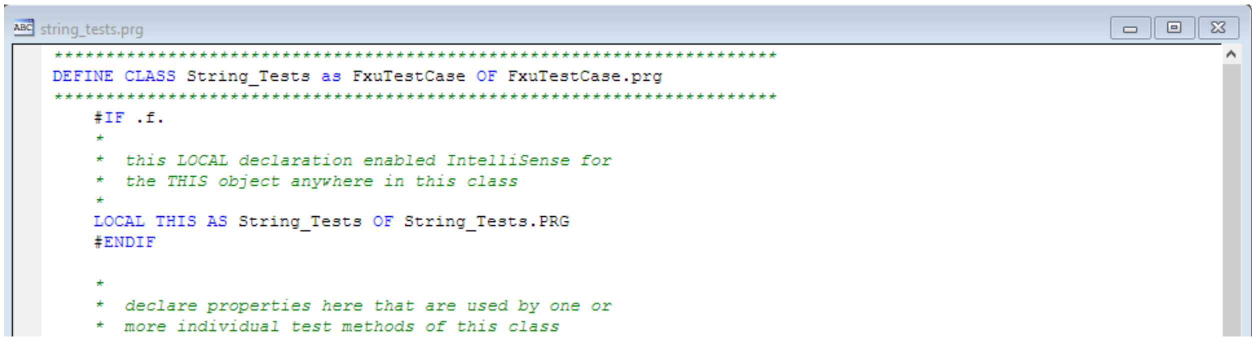
Here you can set a couple of formatting preferences. You can use Beautify on your test class just like any other code of course.

Save As

By default FoxUnit suggests putting your test classes in a *Tests* subfolder of the current folder, which assumes your current folder contains the project or source code you're editing. You can elect to put your tests anywhere you'd like of course, but I do recommend keeping your tests in the same directory structure as your source code.

Lastly you must name your test class, and as I mentioned earlier I recommend naming it *<ClassName>_Tests*. So for our FoxTypes.String class it will be called String_Tests.

Once you click *Create Test Class* you're dropped into the FoxPro code editor with code that is the filled in version of the template you chose.



```
**********************************************************************
DEFINE CLASS String_Tests as FxuTestCase OF FxuTestCase.prg
**********************************************************************
    #IF .f.
    *
    *  this LOCAL declaration enabled IntelliSense for
    *  the THIS object anywhere in this class
    *
    LOCAL THIS AS String_Tests OF String_Tests.PRG
    #ENDIF

    *
    *  declare properties here that are used by one or
    *  more individual test methods of this class
```

Notice that the Test Class itself is a subclass of a class that's built into FoxUnit called FxuTestCase. The class has the following properties by default:

- **ioObjectToBeTested**: FoxUnit test classes are designed to test one of your classes at a time, and this property holds the name of that class. The Setup() method will populate this property once for each test run you execute.

- **icSetClassLib:** This property holds the value of SET CLASSLIB() before you run your tests, and FoxUnit will reset this after each test run. This honors the goal of unit tests not having any side effects.

- **IcTestPrefix:** This property, which defaults to "TEST", indicates the prefix of each of your test methods. If you've ticked the box in the options to honor this property, FoxUnit will only run tests whose names begin with this prefix.

The test class has the following methods that are exposed via the standard template:

- **Setup():** This method is invoked once per test run, regardless of how many unit tests will run. This is where you will instantiate the object you're testing and set the *ioObjectToBeTested* method so it's available from all your unit tests.

- **TearDown():** On the other end is this method, which is invoked after your test run is complete. It restores your environment and releases the *ioObjectToBeTested* instance.

If you add code to either Setup() or TearDown(), you should call DoDefault() to invoke the code in the base class as well.

The *FxuTestCase* has the **MessageOut()** method that was mentioned earlier, which allows you to send status messages to yourself while your tests are running. That's very helpful for times when you want to, say, do performance testing at the same time you're unit testing to see if a tweak you make affects the speed of your function as well as the result.

Assert

The *Assert* methods of the FxuTestCase class are the heart of the unit tests, indicating whether a test succeeded or failed. What defines a success or failure depends on what you were looking for in the test. Usually you'll test that the function/method you're testing returns an expected result. The names of these methods makes their usage self-explanatory:

`AssertEquals`

`AssertEqualsArrays / AssertNotEqualsArrays`

`AssertFalse / AssertTrue`

`AssertHasError`

`AssertHasErrorNo`

`AssertIsObject / AssertIsNotObject`

`AssertNotEmpty`

`AssertNotNull`

`AssertNotNullOrEmpty`

`AssertNotImplemented`

Preparing Our Test Class

Now that we've created our empty test class from the template, here are the steps to get it ready for our unit tests:

1. Uncomment the lines that create the properties we're going to use.
   ```
   ioObjectToBeTested = .NULL.
   icSetClassLib = SPACE(0)
   ```

   Optionally if you're using a test prefix other than the default "TEST", you can set that here as well
   ```
   * icTestPrefix = "<Your preferred prefix here>"
   ```

2. Set those properties in the Setup() method by creating the object, in our case the String class

```
      FUNCTION Setup
         THIS.icSetClassLib = SET("CLASSLIB")
         SET PROCEDURE TO STRING.prg ADDITIVE

         THIS.ioObjectToBeTested = CREATEOBJECT("String")
         ADDPROPERTY(_vfp, "STRING", THIS.ioObjectToBeTested)
```

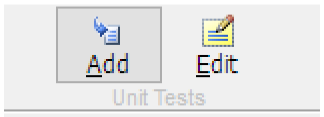
3. Cleanup after yourself in the Teardown() method

```
FUNCTION TearDown
********************************************************************

THIS.ioObjectToBeTested = .NULL.
REMOVEPROPERTY(_Vfp, "STRING")

LOCAL lcSetClassLib
lcSetClassLib = THIS.icSetClassLib
SET CLASSLIB TO &lcSetClassLib
```

Writing Unit Tests

With that preliminary work out of the way, it's time to start writing some actual unit tests. You don't have to save and close the test class to start writing unit tests – you can just start coding them as the next method within the DEFINE…ENDDEFINE of the test class.

If you do close the Test Class screen and then click the Add of the Unit Tests section of the toolbar, FoxUnit will create a "stub" of a unit test for you comments that suggest your next steps and a return value of AssertNotImplemented() to indicate that this unit test has not yet been created. This shows up as a failure on a run because it shows this test still needs attention.

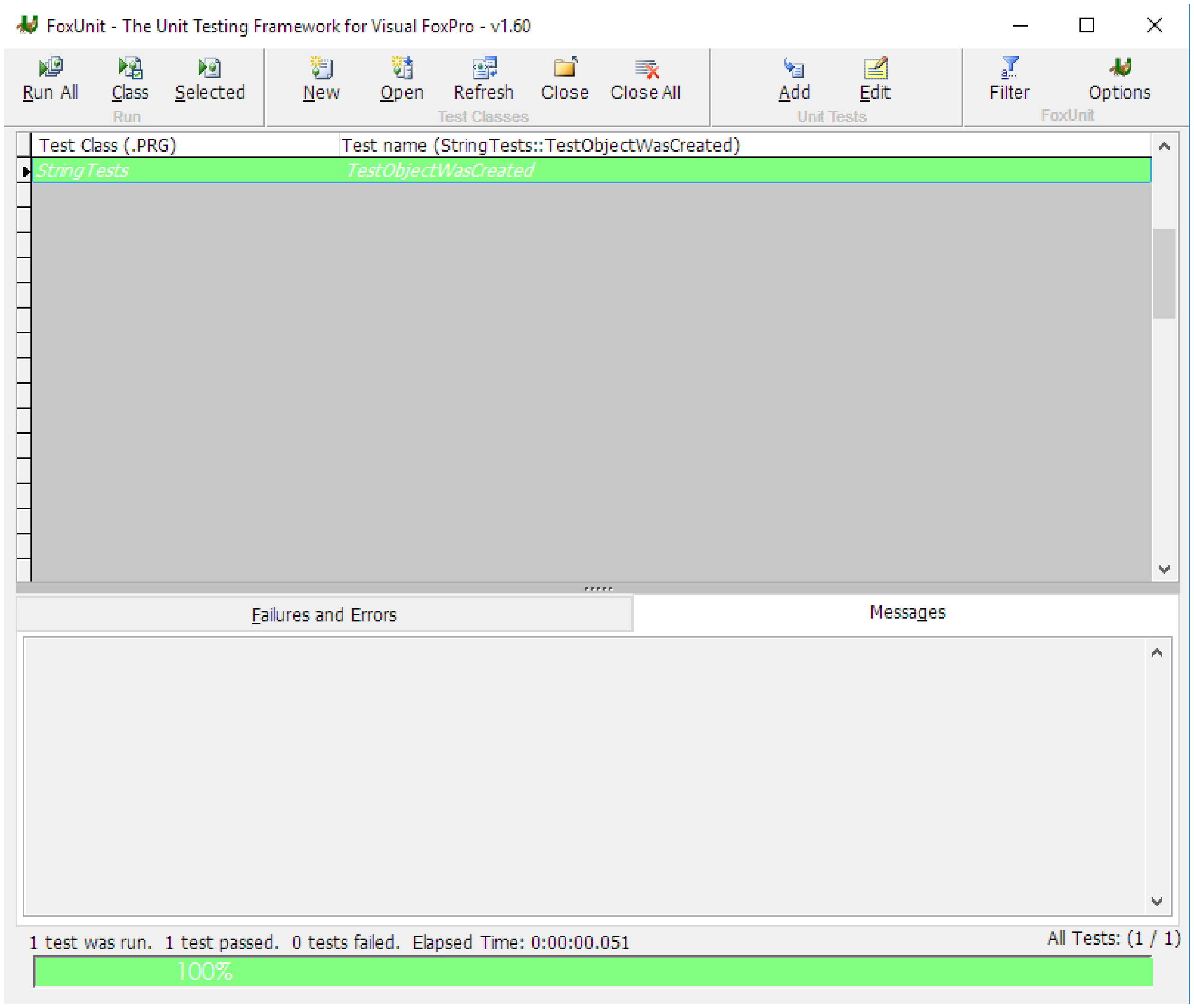


```
  FUNCTION testNewTest
   * 1. Change the name of the test to reflect its purpose. Test one thing only.
   * 2. Implement the test by removing these comments and the default assertion and
writing your own test code.
  RETURN This.AssertNotImplemented()
```

The first unit test I usually create just makes sure that the object I want to test actually got created, so I change the default test to do that:

```
   FUNCTION TestObjectWasCreated
   ********************************************************************
   THIS.AssertNotNull(THIS.ioObjectToBeTested, ;
      "Object was not instantiated during Setup()")
   ********************************************************************
   ENDFUNC
```

Save the test class and Run it and rejoice in the successful test!



Test Class (.PRG)	Test name (StringTests::TestObjectWasCreated)
StringTests	TestObjectWasCreated

1 test was run. 1 test passed. 0 tests failed. Elapsed Time: 0:00:00.051

All Tests: (1 / 1)

100%

If the test failed, read the Failures and Errors message and figure out what went wrong. You can set breakpoints in your unit tests just like any other FoxPro code if you must but keep a couple of things in mind:

1. Breakpoints don't seem to break on FoxPro lines that span across multiple lines.
2. If you have the "Close Debugger" option set in FoxUnit, the debugger will go away when the test run is complete.

More Tests

One of the "tests" we did from the Command Line for our string class looked like this:

```
? _VFP.STRING.Format("Welcome to Southwest Fox {0}!", 2016)
Welcome to Southest Fox 2016
```

As a unit test it looks like this:

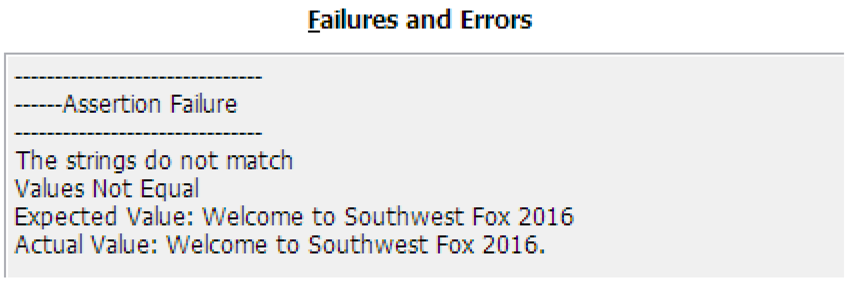
```
   FUNCTION TestStringOneParameter
   cResult = _vfp.String.FORMAT("Welcome to Southwest Fox {0}.", 2016)
   cExpected = "Welcome to Southwest Fox 2016."
   RETURN THIS.AssertEquals(cExpected, cResult "The strings do not match")
   ENDFUNC
```

Notice that I name my unit test with the expected prefix, “Test”, and then the name of the class and then what I’m testing. Your naming convention can vary but this is a pretty standard way of naming your unit tests.

I make a call to the function I’m testing and put it’s value into one variable, *cResult*, and then put the value I’m expecting the function to return into another variable, *cExpected*. Then I use the *THIS.AssertEquals()* method of the test class to compare the values.

The last parameter of all of the Assert() functions is a custom message that gets sent back to the UI console. FoxUnit will automatically echo the Expected and Actual values back to you so you don’t have repeat that information in this parameter.

For example, here’s what I’d get if I had left off the last period of the phrase in c*Expected*:



```
------------------------------
------Assertion Failure
------------------------------
The strings do not match
Values Not Equal
Expected Value: Welcome to Southwest Fox 2016
Actual Value: Welcome to Southwest Fox 2016.
```

Can String.Format handle two parameters? Let’s write a test for it!

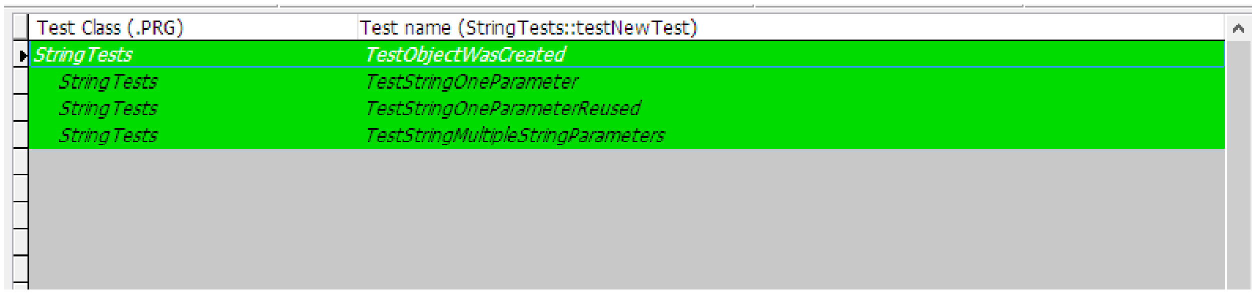
```
   FUNCTION TestStringMultipleStringParameters
   cResult = _vfp.String.FORMAT("The {0} in {1} falls {2} on the {3}.", "rain", 
"Spain", "mainly", "plain")
   cExpected = "The rain in Spain falls mainly on the plain."
   RETURN THIS.AssertEquals(cExpected, cResult, "The strings do not match")
   ENDFUNC
```

(Notice the parameters are zero-based)

What happens if we use the same parameter twice? Let’s write a test for it!

```
   FUNCTION TestStringOneParameterReused
   cResult = _vfp.String.FORMAT("We have nothing to {0} but {0} itself.", "fear")
   cExpected = "We have nothing to fear but fear itself."
   RETURN THIS.AssertEquals(cExpected, cResult, "The strings do not match")
   ENDFUNC
```

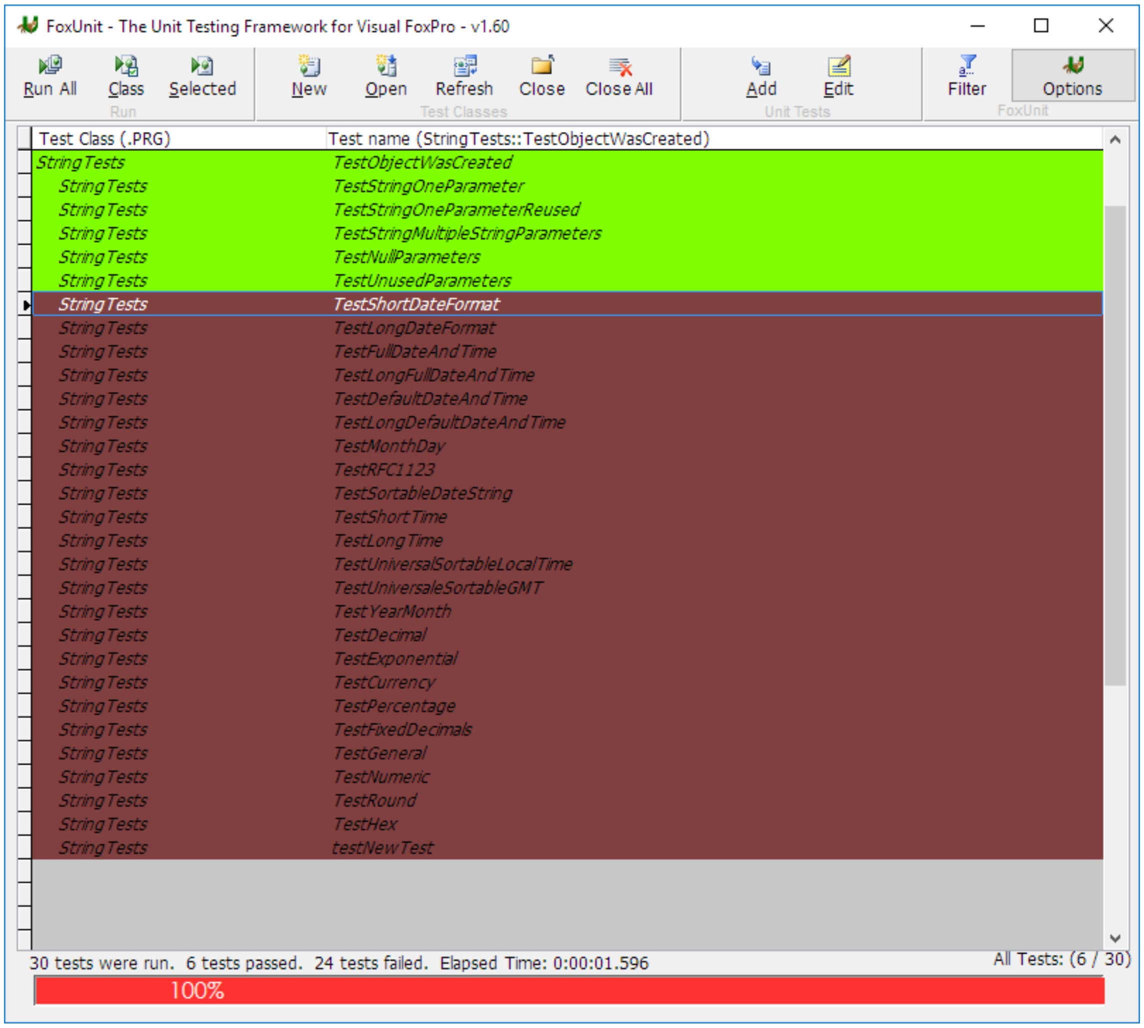
Let’s run all of our tests and see how we’re doing:



Test Class (.PRG)	Test name (StringTests::testNewTest)
StringTests	TestObjectWasCreated
StringTests	TestStringOneParameter
StringTests	TestStringOneParameterReused
StringTests	TestStringMultipleStringParameters

Excellent – Four for Four! Many unit testers speak of the Pavlovian response they get when they see a screenful of successful green tests, and I hope you're beginning to see why.

What else might we want to test? For an existing method like the one we're testing here you'd want to run it throught its paces by throwing unexpected parameters, too many parameters, and parameters of the wrong type. You also want to make sure you test all of the code paths in your method. String.Format method is meant to copy all of the functionality of C#'s String.Format and thus has the ability to convert dates and numbers to strings in a variety of formats. Tests should be written to cover every possibility to ensure



Test Class (.PRG)	Test name (StringTests::TestObjectWasCreated)
StringTests	TestObjectWasCreated
StringTests	TestStringOneParameter
StringTests	TestStringOneParameterReused
StringTests	TestStringMultipleStringParameters
StringTests	TestNullParameters
StringTests	TestUnusedParameters
StringTests	TestShortDateFormat
StringTests	TestLongDateFormat
StringTests	TestFullDateAndTime
StringTests	TestLongFullDateAndTime
StringTests	TestDefaultDateAndTime
StringTests	TestLongDefaultDateAndTime
StringTests	TestMonthDay
StringTests	TestRFC1123
StringTests	TestSortableDateString
StringTests	TestShortTime
StringTests	TestLongTime
StringTests	TestUniversalSortableLocalTime
StringTests	TestUniversaleSortableGMT
StringTests	TestYearMonth
StringTests	TestDecimal
StringTests	TestExponential
StringTests	TestCurrency
StringTests	TestPercentage
StringTests	TestFixedDecimals
StringTests	TestGeneral
StringTests	TestNumeric
StringTests	TestRound
StringTests	TestHex
StringTests	testNewTest

30 tests were run. 6 tests passed. 24 tests failed. Elapsed Time: 0:00:01.596

All Tests: (6 / 30)

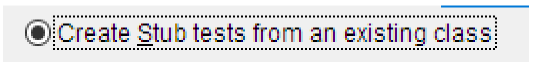
100%

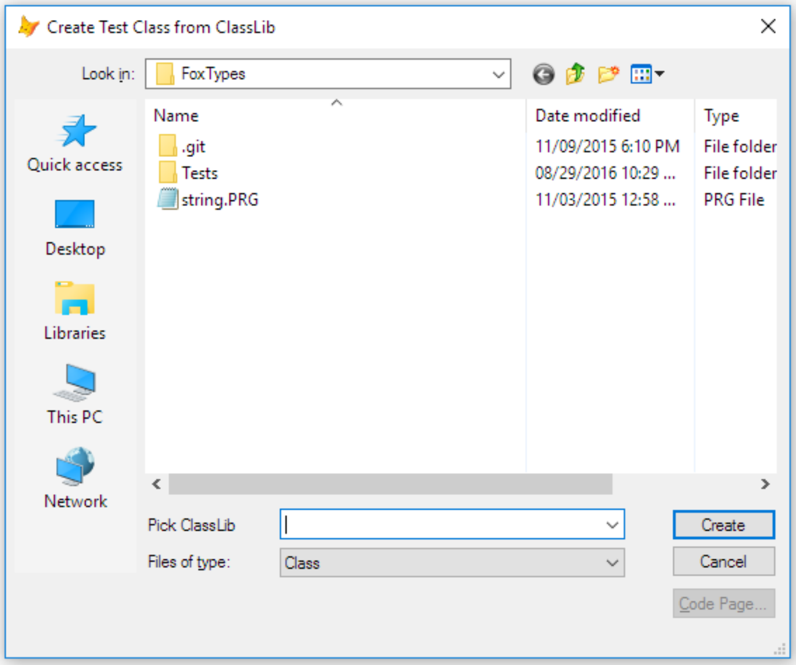
they work correctly. Initially the tests will be "Not Implemented" of course, which are indicates as failures here:

So I encourage you now to go back and write FoxUnit tests for all your existing business objects. A daunting prospect, right? If only there were a way to automatically write tests for all of the methods of a class in a class library.

Well now there is! Version 1.6 adds a new feature (which was actually written about in FoxRockX quite a while ago but is only now making it into the codebase) which lets you choose one of your classes, rather than a template, in order to create basic unit tests. These unit tests will be unimplemented, but they'll at least give you a starting point.

To do this, choose New from the Test Class group of buttons and select this button





Name	Date modified	Type
.git	11/09/2015 6:10 PM	File folder
Tests	08/29/2016 10:29 ...	File folder
string.PRG	11/03/2015 12:58 ...	PRG File

You will be prompted to then select a class library from either a VCX or PRG on disk, and then be presented with a list of all of the classes defined in the class library.

Give your new test class a name, hit Create, and voila! You now have a list of unimplemented unit tests for all of the methods in your class.

Of course you'll have to actually write the tests – this feature isn't *that* smart! But it does give you each class method's signature for making the call which should help somewhat.

And don't forget too that you'll really want to write far more than one test for each method. Really run it through its paces and make sure your code is robust! I don't know if you've seen this online ad but it addresses this very topic:

QA ENGINEER
WALKS INTO A BAR
ORDERS A BEER
ORDERS NULL BEERS
ORDERS 1.33 BEERS
ORDERS A LIZARD
ORDERS -1 BEERS
ORDERS BEERS

Test Driven Development

So far we've written unit tests for existing code, which is critical for ensuring that future changes to that code don't break our programs. "Test Driven Development" flips things around so we write the tests *before*we write the code.

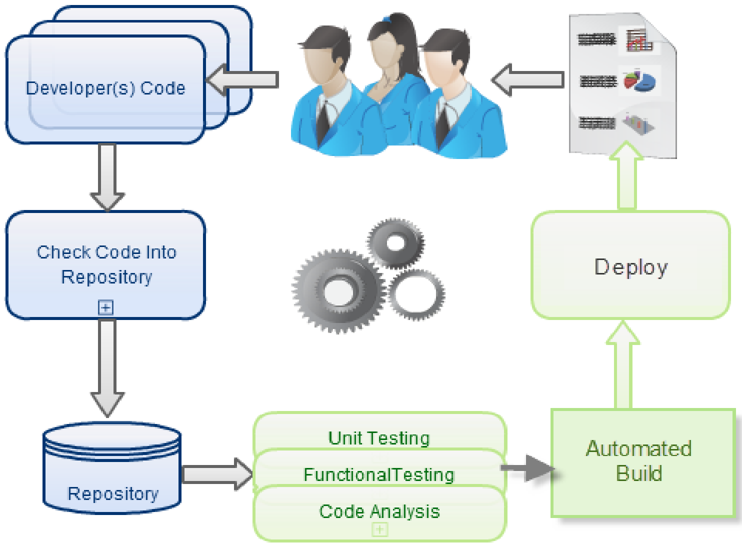
How can you write tests before you write the code you're suppsed to be testing? Won't the tests fail? Absolutely – and that's ok. Writing the tests first make you really think through the design of your class and the implications of those design choices.

Later, as you consider more functionality, you'll get in the habit of first writing the tests and then writing the tests for that new functionality. It takes a while to get used to, and if you're like me you probably won't do it every time religously. Try it out and if you see the benefits of this paradigm then adopt it into your workflow.

Continuous Integration

Unit testing is only one step of the larger software development cycle, which also includes checking out code, making your changes, committing the code, and deploying it. Ideally you want to automate as many of these tasks as you can. Wouldn't it be nice if you could, say, automatically run all of your unit tests when you check in your changes to your source code repository? If anyone's changes "broke" the unit tests, you would get notified, but if everything was copacetic then the changes would automatically get compiled and deployed to the next step? That ideal is known as "continuous integration" and it's made possible using Continuous Integration servers.

For a deeper look at this cycle, check out my Southest Fox 2013 whitepaper "Will the Circle Be Unbroken?"[iv]



The Continuous Integration Cycle

FoxUnit in the Continuous Integration Cycle

In 2014 Fernando Bozzo contributed changes to FoxUnit that allow it to run unit tests from the Command Prompt and generate an artifact that a continuous integration server can parse to determine if the unit tests passed or failed. While I'm not aware of any Continous Integration servers that have prewritten plugins for Visual FoxPro and FoxUnit specifically, they all allow users to write generic plugin scripts. When choosing a Continous Integration server one of the main considerations should be whether you know the scripting language that it uses for the plugins. Another consideration is whether the plugin requires the unit test results to go to "StdOut" (ie the Windows Console) or can be sent to a file. Visual FoxPro does *not* have the ability to write to StdOut, so any CI servers that require that for its pipeline should not be a consideration.

Conclusion

The benefits of writing unit tests should be clearer now. Unit tests not only ensure you haven't broken existing code when making modifications, but if you follow the principles of Test Driven Development, unit tests will clarify your purpose before you begin coding, and direct you once you begin. The unit tests themselves becomes something of a to-do list. And you get to know earlier in the process that things have gone awry which should save you time and your client money.

Clarity and peace of mind are two things we could all use more of, and FoxUnit is a tool to get us there.

I welcome feedback on this whitepaper, and the session itself, as well as any interest you may have in furthering the FoxUnit cause on VFPx. There are many features I'd still love to see added to FoxUnit, such as the ability to watch the filesystem and run tests automatically in the background when a class that has tests gets changed, or the ability to run FoxUnit from the command line. Other platforms' unit testers have these abilities, and we should too! Get in touch with me using the information on the cover page.

Appendix A: FoxTypes Class

```
DEFINE CLASS String AS CUSTOM

   *************************************************************
   FUNCTION Format
   *************************************************************
   * Mimics the String.Format() Method of NET
   *************************************************************
      LPARAMETERS cString, vPara0, vPara1, vPara2, vPara3, vPara4, vPara5, vPara6,
vPara7, vPara8, vPara9
      LOCAL nCount, cCount, cReturn, cSearch, cFormat
      cReturn = cString
      FOR nCount = 1 TO OCCURS("{", cString)
         cSearch = STREXTRACT(cString, "{", "}", nCount, 4)
         cFormat = STREXTRACT(cSearch, ":", "}")
         cCount  = CHRTRAN(STRTRAN(cSearch, cFormat,""), "{:}","")
         IF EMPTY(cFormat)
            cReturn = STRTRAN(cReturn, cSearch,
TRANSFORM(EVALUATE("vPara"+cCount)) )
         ELSE
            xParam = EVALUATE("vPara"+cCount)
            DO CASE
            CASE INLIST(VARTYPE(xParam),'D','T')
               cReturn = STRTRAN(cReturn, cSearch, This.DateFormat(xParam,
cFormat))
            CASE INLIST(VARTYPE(xParam),'N','Y')
               cReturn = STRTRAN(cReturn, cSearch, This.NumericFormat(xParam,
cFormat))
            OTHERWISE
               cReturn = STRTRAN(cReturn, cSearch, TRANSFORM(xParam,cFormat) )
            ENDCASE
         ENDIF
      ENDFOR
      RETURN cReturn

   PROTECTED FUNCTION DateFormat
      LPARAMETERS dtConvert, cFormat
      LOCAL cDate, cCentury, dConvert, cResult
      cResult = ""
      IF VARTYPE(dtConvert)="D"
         dConvert = dtConvert
         dtConvert = DTOT(dConvert)
      ELSE
         dConvert = TTOD(dtConvert)
      ENDIF
      IF LEN(cFormat)=1
         IF INLIST(cFormat, 'r', 'u', 'U')
         * Adjust time to GMT
            DECLARE INTEGER GetTimeZoneInformation IN kernel32 STRING
@lpTimeZoneInformation
            LOCAL cTimeZone, iBiasSeconds
            cTimeZone = REPL(Chr(0), 172)
            GetTimeZoneInformation(@cTimeZone)
            iBiasSeconds = 60 * INT( ASC(SUBSTR(cTimeZone, 1,1)) + ;
                BitLShift(ASC(SUBSTR(cTimeZone, 2,1)),  8) +;
                BitLShift(ASC(SUBSTR(cTimeZone, 3,1)), 16) +;
                BitLShift(ASC(SUBSTR(cTimeZone, 4,1)), 24))
            dtConvert = dtConvert - iBiasSeconds
            dConvert = TTOD(dtConvert)
         ENDIF
         DO CASE
         CASE cFormat='d' && Short date   10/12/2002
            cResult=TRANSFORM(dConvert, "@YS")
         CASE cFormat='D' && Long date    December 10, 2002.  Can't use @YL
            cFormat='MMM dd, yyyy'
         CASE cFormat='f' && Full date & time    December 10, 2002 10:11 PM
            cFormat='MMMM dd, yyyy hh:mm tt'
         CASE cFormat='F' && Full date & time (long)   December 10, 2002 10:11:29 
PM
            cFormat='MMMM dd, yyyy hh:mm:ss tt'
         CASE cFormat='g' && Default date & time    10/12/2002 10:11 PM
            cFormat='dd/MM/yyyy hh:mm tt'
         CASE cFormat='G' && Default date & time (long)    10/12/2002 10:11:29 PM
            cFormat='dd/MM/yyyy hh:mm tt'
         CASE cFormat='M' && Month day pattern   December 10
            cFormat='MMMM dd'
         CASE cFormat='r' && RFC1123 date string    Tue, 10 Dec 2002 22:11:29 GMT
            cFormat='ddd, dd MMM yyyy hh:mm:ss GMT'
         CASE cFormat='s' && Sortable date string   2002-12-10T22:11:29
            cResult = TTOC(dtConvert,3)
         CASE cFormat='t' && Short time   10:11 PM
            cFormat='hh:mm tt'
         CASE cFormat='T' && Long time    10:11:29 PM
            cFormat='hh:mm:ss tt'
         CASE cFormat='u' && Universal sortable, local time   2002-12-10 22:13:50Z
            cFormat='yyyy-MM-dd hh:mm:ddZ'
         CASE cFormat='U' && Universal sortable, GMT   December 11, 2002 3:13:50 
AM
            cFormat="MMMM dd, yyyy hh:mm:ss tt"
         CASE cFormat='Y' && Year month pattern  December, 2002
            cFormat="MMMM, yyyy"
         ENDCASE
      ENDIF
      IF EMPTY(cResult) AND LEN(cFormat)>1
         cResult=This.ParseDateFormat(cFormat, dtConvert)
      ENDIF
      RETURN cResult


   PROTECTED FUNCTION ParseDateFormat
      LPARAMETERS cFormat, dtVar
      cFormat=STRTRAN(cFormat,"hh", PADL(HOUR(dtVar),2,'0'))
      cFormat=STRTRAN(cFormat,"mm", PADL(MINUTE(dtVar),2,'0'))
      cFormat=STRTRAN(cFormat,"ss", PADL(SEC(dtVar),2,'0'))
      cFormat=STRTRAN(cFormat,"MMMM", CMONTH(dtVar))
      cFormat=STRTRAN(cFormat,"MMM", LEFT(CMONTH(dtVar),3))
      cFormat=STRTRAN(cFormat,"MM", PADL(MONTH(dtVar),2,'0'))
      cFormat=STRTRAN(cFormat,"ddd", LEFT(CDOW(dtVar),3))
      cFormat=STRTRAN(cFormat,"dd", PADL(DAY(dtVar),2,'0'))
      cFormat=STRTRAN(cFormat,"yyyy", TRANSFORM(YEAR(dtVar)))
      cFormat=STRTRAN(cFormat,"yy", RIGHT(TRANSFORM(YEAR(dtVar)),2))
      cFormat=STRTRAN(cFormat,"tt", IIF(HOUR(dtVar)<12,"AM","PM"))
      RETURN cFormat

   PROTECTED FUNCTION NumericFormat
      LPARAMETERS nConvert, cFormatCode
      LOCAL cResult, cFormat
      cResult = ""
      cFormat = UPPER(LEFT(cFormatCode,1))
      iWidth = VAL(SUBSTR(cFormatCode,2))
      DO CASE
         CASE cFormat='D' AND nConvert=INT(nConvert)    && Decimal
            cResult=TRANSFORM(nConvert)
         CASE cFormat='E' && Exponential
            cResult=TRANSFORM(nConvert, "@^")
         CASE INLIST(cFormat,'C','F','P') && Fixed # of decimal place (default 0)
            IF cFormat='P'
               nConvert = nConvert * 100
            ENDIF
            cResult=ALLTRIM(TRANSFORM(nConvert,
REPLICATE('9',12)+IIF(iWidth<=0,'','.'+REPLICATE('9',iWidth))))
            IF cFormat='C'
               cResult = '$'+cResult
            ELSE
               IF cFormat='P'
                  cResult = cResult + '%'
               ENDIF
            ENDIF

         CASE cFormat='G' && General
            cResult=TRANSFORM(nConvert)
         CASE cFormat='N' && Numeric
            cResult=TRANSFORM(nConvert)
         CASE cFormat='P' && Percent
            iWidth = IIF(LEN(cFormat)=1,2,iWidth)  && Default to 2 decimal places
            cResult= TRANSFORM(nConvert*100) +'%'
         CASE cFormat='R' && Round
            cResult=TRANSFORM(nConvert)
         CASE cFormat='X' && Hex
            cResult=TRANSFORM(nConvert, "@0")
      ENDCASE

      RETURN cResult



ENDDEFINE
```

Appendix B – The Standard FoxUnit Test Case Template

```
**********************************************************************
DEFINE CLASS <<testclass>> as FxuTestCase OF FxuTestCase.prg
**********************************************************************
   #IF .f.
   *
   *  this LOCAL declaration enabled IntelliSense for
   *  the THIS object anywhere in this class
   *
   LOCAL THIS AS <<testclass>> OF <<testclass>>.PRG
   #ENDIF

   *
   *  declare properties here that are used by one or
   *  more individual test methods of this class
   *
   *  for example, if you create an object to a custom
   *  THIS.Property in THIS.Setup(), estabish the property
   *  here, where it will be available (to IntelliSense)
   *  throughout:
   *
*!*    ioObjectToBeTested = .NULL.
*!*    icSetClassLib = SPACE(0)


   * the icTestPrefix property in the base FxuTestCase class defaults
   * to "TEST" (not case sensitive). There is a setting on the interface
   * tab of the options form (accessible via right-clicking on the
   * main FoxUnit form and choosing the options item) labeld as
   * "Load and run only tests with the specified icTestPrefix value in test
classes"
   *
   * If this is checked, then only tests in any test class that start with the
   * prefix specified with the icTestPrefix property value will be loaded
   * into FoxUnit and run. You can override this prefix on a per-class basis.
   *
   * This makes it possible to create ancillary methods in your test classes
   * that can be shared amongst other test methods without being run as
   * tests themselves. Additionally, this means you can quickly and easily
   * disable a test by modifying it and changing it's test prefix from
   * that specified by the icTestPrefix property

   * Additionally, you could set this in the INIT() method of your derived class
   * but make sure you dodefault() first. When the option to run only
   * tests with the icTestPrefix specified is checked in the options form,
   * the test classes are actually all instantiated individually to pull
   * the icTestPrefix value.

*!*    icTestPrefix = "<Your preferred prefix here>"

   ********************************************************************
   FUNCTION Setup
   ********************************************************************
   *
   *  put common setup code here -- this method is called
   *  whenever THIS.Run() (inherited method) to run each
   *  of the custom test methods you add, specific test
   *  methods that are not inherited from FoxUnit
   *
   *  do NOT call THIS.Assert..() methods here -- this is
   *  NOT a test method
   *
    *  for example, you can instantiate all the object(s)
    *  you will be testing by the custom test methods of
    *  this class:
*!*    THIS.icSetClassLib = SET("CLASSLIB")
*!*    SET CLASSLIB TO MyApplicationClassLib.VCX ADDITIVE
*!*    THIS.ioObjectToBeTested = CREATEOBJECT("MyNewClassImWriting")

   ********************************************************************
   ENDFUNC
   ********************************************************************

   ********************************************************************
   FUNCTION TearDown
   ********************************************************************
   *
   *  put common cleanup code here -- this method is called
   *  whenever THIS.Run() (inherited method) to run each
   *  of the custom test methods you add, specific test
   *  methods that are not inherited from FoxUnit
   *
   *  do NOT call THIS.Assert..() methods here -- this is
   *  NOT a test method
   *
    *  for example, you can release  all the object(s)
    *  you will be testing by the custom test methods of
    *  this class:
*!*     THIS.ioObjectToBeTested = .NULL.
*!*    LOCAL lcSetClassLib
*!*    lcSetClassLib = THIS.icSetClassLib
*!*    SET CLASSLIB TO &lcSetClassLib

   ********************************************************************
   ENDFUNC
   ********************************************************************


   *
   *  test methods can use any method name not already used by
   *  the parent FXUTestCase class
   *    MODIFY COMMAND FXUTestCase
   *  DO NOT override any test methods except for the abstract
   *  test methods Setup() and TearDown(), as described above
   *
   *  the three important inherited methods that you call
   *  from your test methods are:
   *    THIS.AssertTrue(<Expression>,"Failure message")
   *    THIS.AssertEquals(<ExpectedValue>,<Expression>,"Failure message")
   *    THIS.AssertNotNull(<Expression>,"Failure message")
   *  all test methods either pass or fail -- the assertions
   *  either succeed or fail

   *
   *  here's a simple AssertNotNull example test method
   *
*!*    ********************************************************************
*!*    FUNCTION TestObjectWasCreated
*!*    ********************************************************************
*!*    THIS.AssertNotNull(THIS.ioObjectToBeTested, ;
*!*       "Object was not instantiated during Setup()")
*!*    ********************************************************************
*!*    ENDFUNC
*!*    ********************************************************************

   *
   *  here's one for AssertTrue
   *
*!*    ********************************************************************
*!*    FUNCTION TestObjectCustomMethod
*!*    ********************************************************************
*!*    THIS.AssertTrue(THIS.ioObjectToBeTested.CustomMethod()), ;
          "Object.CustomMethod() failed")
*!*    ********************************************************************
*!*    ENDFUNC
*!*    ********************************************************************

   *
   *  and one for AssertEquals
   *
*!*    ********************************************************************
*!*    FUNCTION TestObjectCustomMethod100ReturnValue
*!*    ********************************************************************
*!*
*!*    * Please note that string Comparisons with AssertEquals are
*!*    * case sensitive.
*!*
*!*    THIS.AssertEquals("John Smith", ;
*!*                   THIS.ioObjectToBeTested.Object.CustomMethod100(), ;
*!*                   "Object.CustomMethod100() did not return 'John Smith'",
*!*    ********************************************************************
*!*    ENDFUNC
*!*    ********************************************************************


**********************************************************************
ENDDEFINE
**********************************************************************
```

Appendix C – Minimal FoxUnit Test Case Template

```
**********************************************************************
DEFINE CLASS <<testclass>> as FxuTestCase OF FxuTestCase.prg
**********************************************************************

   #IF .f.
   LOCAL THIS AS <<testclass>> OF <<testclass>>.PRG
   #ENDIF

   ********************************************************************
   FUNCTION Setup
   ********************************************************************

   ********************************************************************
   ENDFUNC
   ********************************************************************

   ********************************************************************
   FUNCTION TearDown
   ********************************************************************

   ********************************************************************
   ENDFUNC
   ********************************************************************


**********************************************************************
ENDDEFINE
**********************************************************************
```

[i] http://west-wind.com/WestwindClientTools.aspx

[ii] http://www.tomorrowssolutionsllc.com/Conference%20Sessions/Getting%20Your%20Head%20Around%20Business%20Objects.pdf

[iii] FoxUnit 1.6 will be released around the same time as Southwest Fox 2016

[iv] http://saltydogllc.com/wp-content/uploads/SELJE_Continuous-Integration-with-VFP.pdf