VBA Rubberducking (Part 3)

Posted on May 18, 2016May 18, 2016 by Rubberduck VBA
This post is the third in a series of post that walk you through the various features of the Rubberduck open-source VBE add-in.

- Part 1 (https://rubberduckvba.wordpress.com/2016/05/04/vba-rubberducking-part-1/) introduced the **navigation** features.
- Part 2 (https://rubberduckvba.wordpress.com/2016/05/14/vba-rubberducking-part-2/) covered the **code inspections**.

Unit Testing

If you've been following Rubberduck since its early days, you already know that this is where and how the project started. Before Rubberduck was a VBE add-in, it was an Excel add-in completely written in VBA, that started with this Code Review post (http://codereview.stackexchange.com/q/62781/23788); before Rubberduck was even *named* "Rubberduck", it was a C# port of this VBA code (http://codereview.stackexchange.com/q/63004/23788) – the idea being to enable writing and running unit tests beyond Excel, in Access and Word VBA as well, without having to replicate all that code in multiple add-in projects.

Zero Boilerplate

There *are* other VBA unit testing solutions out there. A lot require quite a bit of boilerplate setup code; those written in VBA require programmatic access to the VBIDE object model, which may be a security concern (you're allowing VBA to execute code that can *generate and run* VBA code after all). Rubberduck unit tests require neither. Because it's a VBE add-in, Rubberduck *already* has programmatic access to the code in the IDE, and the ability to scan, modify, generate and execute VBA code – without requiring a dent in your corporate security policy.

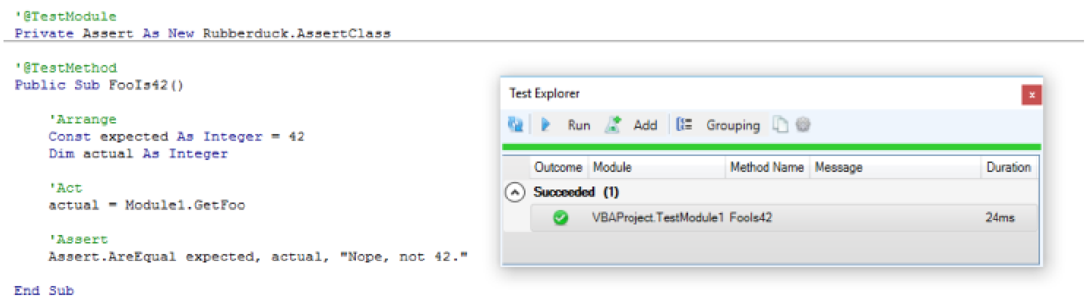
Rubberduck requires pretty much zero boilerplate. This is a fully working test module:

```
'@TestModule
Private Assert As Rubberduck.AssertClass

'@TestMethod
Public Sub FooIs42()

    'Arrange
    Const expected As Integer = 42
    Dim actual As Integer

    'Act
    actual = Module1.GetFoo

    'Assert
    Assert.AreEqual expected, actual, "Nope, not 42."

End Sub
```

Okay, it's just an example. But still, it shows how little is *required* for it to work:

- A **@TestModule** annotation in the *declarations section* of a standard module.
- A **Rubberduck.AssertClass** instance, which can be late or early-bound.
- A **@TestMethod** annotation to formally identify a test method.

That's all. And up until recently, the **@TestMethod** annotation was optional – in Rubberduck 1.x, if you had a public parameterless method with a name that starts with "Test", in a standard module, Rubberduck treated it as a test method. **This is changing in 2.0**, as we are making the **@TestMethod** annotation mandatory, favoring explicitness over implicit naming conventions. Test methods still need to be public and parameterless, and in a standard module though.



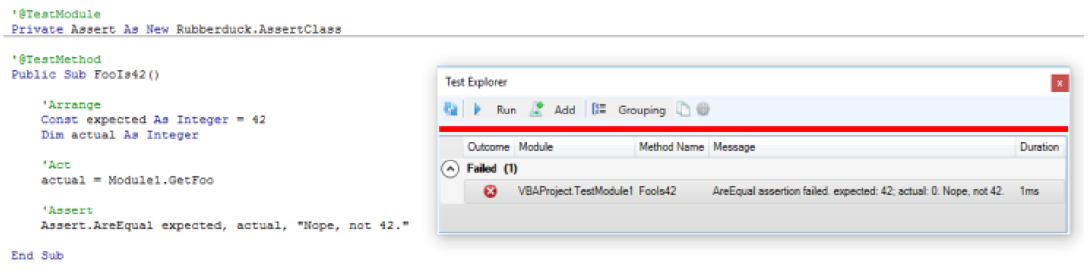
```
'@TestModule
Private Assert As New Rubberduck.AssertClass

'@TestMethod
Public Sub FooIs42()

    'Arrange
    Const expected As Integer = 42
    Dim actual As Integer

    'Act
    actual = Module1.GetFoo

    'Assert
    Assert.AreEqual expected, actual, "Nope, not 42."

End Sub
```

Test Explorer

Run | Add | Grouping

Outcome	Module	Method Name	Message	Duration
Succeeded (1)				
✓	VBAProject.TestModule1	FooIs42		24ms

Now, let's say **GetFoo** returning 42 is a business requirement, and that something needs to change in **Module1** or elsewhere and, inadvertently, **GetFoo** starts returning 0. If you don't have a unit test that documents and verifies that business requirement, you've introduced a bug that may take a while to be discovered. However if you *do* have a test for it, and that you've made it a habit to run your test suite whenever you make a change *just to be sure* that all the business requirements are still met…



```
'@TestModule
Private Assert As New Rubberduck.AssertClass

'@TestMethod
Public Sub FooIs42()

    'Arrange
    Const expected As Integer = 42
    Dim actual As Integer

    'Act
    actual = Module1.GetFoo

    'Assert
    Assert.AreEqual expected, actual, "Nope, not 42."

End Sub
```

Test Explorer

Run | Add | Grouping

Outcome	Module	Method Name	Message	Duration
Failed (1)				
✗	VBAProject.TestModule1	FooIs42	AreEqual assertion failed. expected: 42; actual: 0. Nope, not 42.	1ms

Then you have a failing test, and **you know right away** that your modification has subtly introduced a change in behavior that *will* be reported as a bug sooner or later.

If you've already written unit tests, I'm probably preaching to the choir here. If you've only ever written VBA code, it's possible you've *heard* of unit testing before, but aren't quite sure how you could make *your code* work with it.

Luckily, the key concepts are language-agnostic, and **VBA definitely has support for everything you need for full-blown Test-Driven Development**.

Code Against Abstractions

Whether you're writing C#, Java, PHP, Python, Ruby, or VBA, if your code is *tightly coupled* with a UI, accessing the file system, a Web service, a database, …or a worksheet, then it's not *fit* for a unit test, because a *unit* test…

- Should be fast
- Should not have side-effects
- Should not depend on (or impact) other tests
- Should have all dependencies under control
- Should test *one thing*, and have *one reason to fail*

Wait. My code *is* accessing a worksheet. Does that mean I can't write tests for it?

Yes and no. I'll tell you a secret. Quite a lot of VBA posts I see on Code Review are asking for tips to get their code to run faster with large data sets. Something I often say in my reviews, is that **the single slowest thing you can do in VBA is access a worksheet**.

Don't code your logic against the worksheet, code your logic against *an abstraction* of the worksheet. An array is often all you need: refactor your logic to work with an array instead of a worksheet, and not only you'll be able to write a test that gives it any array you want, your code will also perform better!

Encapsulate your logic in class modules, test the public interface; if the logic brings up a UI (even a message box!), *extract that piece of code elsewhere* – make it the responsibility of *something else*, get it out of the way so your tests can concentrate on the actual important things that they're testing for.

A whole book could be written about *reducing coupling* in code, *increasing cohesion*, and writing tests in general. Poke around, research a bit. You'll see where Rubberduck wants to take your VBA code.

The Test Explorer

Rubberduck's *Test Explorer* offers two main "sets" of commands: "Run", and "Add".

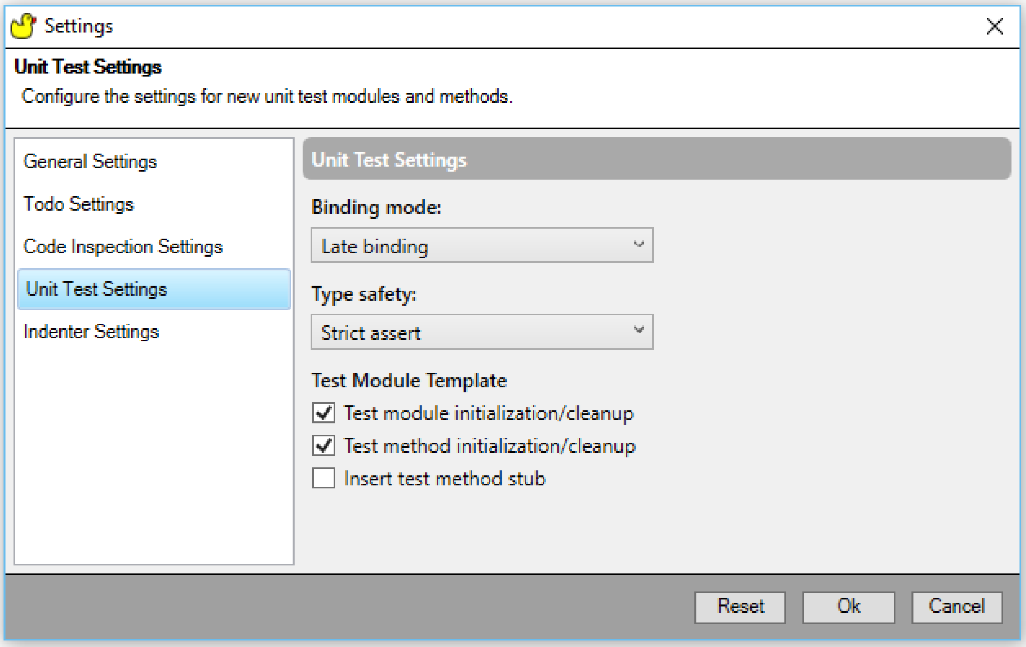
The "Add" menu lets you easily add a **test module** to your project, and from there you can just as easily add a **test method**, one of two templates:

- **Test Method** is the standard Arrange-Act-Assert deal, with error handling that ensures the test will correctly fail on error *and report* that error.
- **Test Method (Expected Error)** is the same AAA deal, except this template is for writing tests that are *expected* to raise a specific runtime error; such tests *fail* if the expected error isn't raised.

The "Run" menu lets you easily run all, or a subset of the tests – e.g. you might want to only run the tests that failed the last time you ran them.

Results can be regrouped either by outcome or by location (project/module), and again can be copied to the clipboard with a single click.

Test settings let you control the contents of the test module template:


Settings

Unit Test Settings

Configure the settings for new unit test modules and methods.

Unit Test Settings

Binding mode: Late binding

Type safety: Strict assert

Test Module Template

- [x] Test module initialization/cleanup
- [x] Test method initialization/cleanup
- [ ] Insert test method stub

Binding mode determines whether the **AssertClass** instance is going to be declared "As Object" (late-bound, default) or "As New Rubberduck.AssertClass" (early-bound).

Type safety determines whether the **Assert** variable is going to be a Rubberduck.AssertClass (strict) or a Rubberduck.PermissiveAssertClass (permissive); the permissive asserts differs with the strict (original and default) version in that equality checks are more closely modeled on VBA equality rules: with a permissive assert, an Integer value of 254 can be compared to a Byte value of 254 and deemed equal. Strict equality requires the types to match, not just the value.

Test Module Template checkboxes determine whether the **@TestInitialize**, **@TestCleanup**, **@ModuleInitialize** and **@ModuleCleanup** method stubs are going to be generated, and also whether creating a new test module creates a test method by default.

All these settings only affect *new* test modules, not existing ones.

The Assert Class

Tests *assert* things. Without assertions, a Rubberduck test can't have a meaningful result, and will simply pass. The **IAssert** interface (implemented by both **AssertClass** and **PermissiveAssertClass**) exposes a number of members largely inspired by MS-Tests in Visual Studio:

Name	Description
AreEqual	Verifies that two specified objects are equal. The assertion fails if the objects are not equal.
AreNotEqual	Verifies that two specified objects are not equal. The assertion fails if the objects are equal.
AreNotSame	Verifies that two specified object variables refer to different objects. The assertion fails if they refer to the same object.
AreSame	Verifies that two specified object variables refer to the same object. The assertion fails if they refer to different objects.
Fail	Fails the assertion without checking any conditions.
Inconclusive	Indicates that the assertion cannot be verified.
IsFalse	Verifies that the specified condition is false. The assertion fails if the condition is true.
IsNothing	Verifies that the specified object is Nothing. The assertion fails if it is notNothing.
IsNotNothing	Verifies that the specified object is not Nothing. The assertion fails if it isNothing.
IsTrue	Verifies that the specified condition is true. The assertion fails if the condition is false.

To be continued…

AUTOMATTIC

You don't need to go to an office to write code. Work with us!

APPLY

REPORT THIS AD

Posted in OOP, open-source, rubberduck, vbaTagged add-in, rubberduck, unit-testing, v2.0, vba, vba tools, vbe

Published by Rubberduck VBA

I'm Mathieu Guindon (Microsoft MVP Office Apps & Services, 2018), you may have known me as "Mat's Mug" on Stack Overflow and Code Review Stack Exchange. I manage the Rubberduck open-source project, whose goal is to bring the Visual Basic Editor (VBE) - VBA's IDE - into the 21st century, by providing features modern IDE's provide. *View all posts by Rubberduck VBA*

1 thought on "VBA Rubberducking (Part 3)"

1. **VBA Rubberducking (Part 4) – Rubberduck News** *May 28, 2016 Reply*

[…] Part 3 featured the unit testing feature. […]