How to unit test VBA code?

Posted on October 19, 2017March 28, 2018 by Rubberduck VBA

So Rubberduck lets you write unit tests for your VBA code. If you're learning VBA, or if you're a seasoned VBA developer but have never written a unit test before, this can sound like a crazy useless idea. I know, because before I started working on Rubberduck, that's how I was seeing unit tests: annoying, redundant code that tells you nothing F5/debugging doesn't already tell you.

Right? What's the point anyway?

First, it changes how you *think* about code. Things like the *Single Responsibility Principle* start becoming freakishly important, and you begin to break that monolithic macro procedure into smaller, more *focused* chunks. *Future you*, or whoever inherits your code, will be extremely thankful for that.

But not all VBA code *should* be unit-tested. Let's see why.

Know what NOT to test

All code has *dependencies*. Some of these dependencies we can't really do without, and don't really affect anything – global-scope functions in the VBA Standard Library, for example. Other dependencies affect global state, require user input (MsgBox, InputBox, forms, dialogs, etc.) or access external resources – like a database, the file system, …or a worksheet.

For the sake of this article, say you have a simple procedure like this:

```
Public Sub DoSomething()
    Dim conn As ADODB.Connection
    Set conn = New ADODB.Connection
    conn.ConnectionString = "{connection string}"
    conn.Open
    Dim rs As ADODB.Recordset
    Set rs = conn.Execute("SELECT * FROM dbo.SomeTable")
    Sheet1.Range("A1").CopyFromRecordset rs
    conn.Close
End Sub
```

The problematic dependencies are:

- `conn`, an ADODB connection
- `rs`, an ADODB recordset
- `Sheet1`, an Excel worksheet

Is that procedure doomed and completely untestable? Well, as is, …pretty much: the only way to write a test for this procedure would be to *actually* run it, and verify that something was dumped into `Sheet1`. In fact, that's pretty much automating F5-debug: it's an *integration test*, not a *unit test* – it's a test, but it's validating that all components work together. It's not useless, but that's not a unit test.

Refactoring

The procedure needs to be parameterless, because it's invoked from some button: so we have a major problem here – there's no way to factor out the dependencies!

Or is there? What if we introduced a class, and moved the functionality into there?

Now we'd be looking at this:

```
Public Sub DoSomething()
    With New MyTestableMacro
        .Run
    End With
End Sub
```

At this point we tremendously increased the macro's *abstraction level* and that's awesome, but we didn't really gain anything. Or did we? Now that we've *decoupled* the macro's entry point from the implementation, we can pull out the dependencies and unit-test the `MyTestableMacro` class! But how do we do that?

Think in terms of *concerns*:

- Pulling data from a database
- Writing the data to a worksheet

Now think in terms of *objects*:

- We need some *data service* responsible for *pulling data from a database*
- We need some *spreadsheet service* responsible for *writing data to a worksheet*

The macro might look like this now:

```
Public Sub DoSomething()

    Dim dataService As MyDbDataService
    Set dataService = New MyDbDataService

    Dim spreadsheetService As Sheet1Service
    Set spreadsheetService = New Sheet1Service

    With New MyTestableMacro
        .Run dataService, spreadsheetService
    End With

End Sub
```

Now if we think of `MyDbDataService` as an *interface*, we could conceptualize it like this:

```vb
Option Explicit
'@Folder "Services.Abstract"
'@Interface IDataService

Public Function GetSomeTable() As Variant
End Function
```

And if we think of `Sheet1Service` as an *interface*, we could conceptualize it like this:

```vb
Option Explicit
'@Folder "Services.Abstract"
'@Interface IWorksheetService

Public Sub WriteAllData(ByRef data As Variant)
End Sub
```

Notice the interfaces don't know or care about `ADODB.Recordset`: the last thing we want is to have that dependency in our way, so we'll be passing a `Variant` array around instead of a recordset.

Now the `Run` method's signature might look like this:

```vb
Public Sub Run(ByVal dataService As IDataService, ByVal wsService As IWorksh
```

Notice it only knows about *abstractions*, not the concrete implementations. All that's missing is to make `MyDbDataService` implement the `IDataService` interface, and `Sheet1Service` implement the `IWorksheetService` interface.

```vb
Option Explicit
Implements IDataService
'@Folder "Services.Concrete"

Private Function IDataService_GetSomeTable() As Variant
    Dim conn As ADODB.Connection
    Set conn = New ADODB.Connection
    conn.ConnectionString = "{connection string}"
    conn.Open
    Dim rs As ADODB.Recordset
    Set rs = conn.Execute("SELECT * FROM dbo.SomeTable")
    'dump the recordset onto a temp sheet:
    Dim tempSheet As Excel.Worksheet
    Set tempSheet = ThisWorkbook.Worksheets.Add
    tempSheet.Range("A1").CopyFromRecordset rs
    IDataService_GetSomeTable = tempSheet.UsedRange.Value '2D variant array
    conn.Close
    tempSheet.Delete
End Function
```

Stubbing the interfaces

So here's where the magic begins: the macro will definitely be using the above implementation, but *nothing forces a unit test to use it too*. A unit test would be happy to use something like this:

```vb
Option Explicit
Implements IDataService
'@Folder "Services.Stubs"

Private Function IDataService_GetSomeTable() As Variant
    Dim result(1 To 50, 1 To 10) As Variant
    IDataService_GetSomeTable = result
End Function

Public Function GetSomeTable() As Variant
    GetSomeTable = IDataService_GetSomeTable
End Function
```

You could populate the array with some fake results, expose properties and methods to configure the stub in every way your tests require (depending on what logic needs to run against the data after it's dumped onto the worksheet) – for this example though all we need is for the method to return a 2D variant array, and the above code satisfies that.

Then we need a stub for the `IWorksheetService` interface, too:

```vb
Option Explicit
Implements IWorksheetService
'@Folder "Services.Stubs"

Private written As Boolean
Private arrayPointer As Long

Private Sub IWorksheetService_WriteAllData(ByRef data As Variant)
    written = True
    arrayPointer = VarPtr(data)
End Function

Public Property Get DataWasWritten() As Boolean
    DataWasWritten = written
End Property

Public Property Get WrittenArrayPointer() As Long
    WrittenArrayPointer = arrayPointer
End Property
```

Writing the tests

That's all our test needs for now. See where this is going? `DoSomething` is using concrete implementations of the service interfaces that *actually do the work*, and a unit test can look like this:

```vba
'@TestMethod
Public Sub GivenData_WritesToWorksheet()
    'Arrange
    Dim dataServiceStub As MyDataServiceStub
    Set dataServiceStub = New MyDataServiceStub
    Dim wsServiceStub As MyWorksheetServiceStub
    Set wsServiceStub = New MyWorksheetServiceStub

    'Act
    With New MyTestableMacro
        .Run dataServiceStub, wsServiceStub
    End With

    'Assert
    Assert.IsTrue wsServiceStub.DataWasWritten
End Sub
```

If `MyTestableMacro.Run` invokes `IWorksheetService.WriteAllData`, this test will pass.

One more:

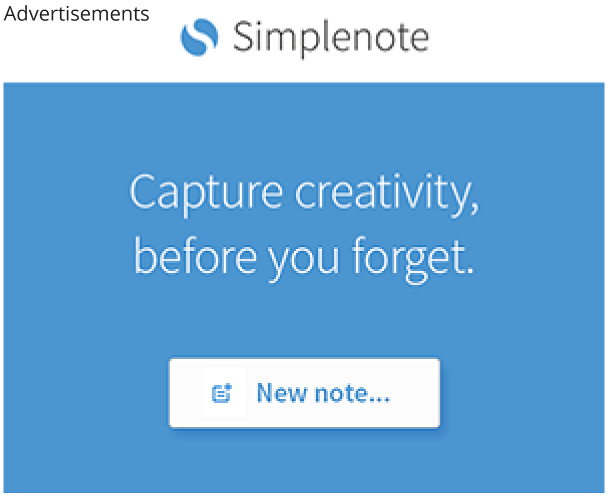
```vba
'@TestMethod
Public Sub WorksheetServiceWorksOffDataFromDataService()
    'Arrange
    Dim dataServiceStub As MyDataServiceStub
    Set dataServiceStub = New MyDataServiceStub
    Dim expected As Long
    expected = VarPtr(dataServiceStub.GetSomeTable)

    Dim wsServiceStub As MyWorksheetServiceStub
    Set wsServiceStub = New MyWorksheetServiceStub

    'Act
    With New MyTestableMacro
        .Run dataServiceStub, wsServiceStub
    End With

    Dim actual As Long
    actual = wsServiceStub.WrittenArrayPointer

    'Assert
    Assert.AreEqual expected, actual
End Sub
```

If the worksheet service receives the exact same array that the data service returned, this test should pass.

That was a relatively trivial example – the overhead (5 classes, including 2 interfaces and 2 stub implementations) is probably not justified given the simplicity of the task at hand (pull data from a database, dump that data to a worksheet). But hopefully it illustrates a number of things:

- How to pull dependencies out of the logic that needs to be tested.
- How to abstract the dependencies as interfaces.
- How to implement test stubs for these dependencies, and how stubs can expose members that aren't on the interface, for the tests to consume.
- How unit tests document what the code is supposed to be doing, through descriptive naming.
- VBA code can be just as object-oriented as any other code, with full-blown *polymorphism* and *dependency injection*.

Next tutorial should be about `MSForms.UserForm`, how *not* to use it, and how to test code that needs to pop a dialog. I didn't mention anything about Rubberduck's `Fakes` framework here either, but know that if one of your dependencies is a `MsgBox` and you have different code paths depending on whether the user clicked [Ok] or [Cancel], you can use Rubberduck's `Fakes` API to *literally* configure how the `MsgBox` statement is going to behave when it's invoked by a Rubberduck test.



Posted in *[OOP](), [tutorials](), [unit-testing](), [vba]()Tagged [oop](), [unit-testing](), [vba]()*

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]()*

9 thoughts on “How to unit test VBA code?”

1. **Manju** *December 18, 2017* *Reply*

   rubber duck unit tests can be possible to run on jenkins”?
   or Is there any way can run on jenkins server

   Rubberduck VBA *December 18, 2017* *Reply*

   VBA code is executed by the VBE, not by Rubberduck – we parse the code, we don’t interpret it. And running a VBA host application on a server isn’t recommended. So, no.

2. **Manju** *December 22, 2017* *Reply*

   to make it clear, my intention is to write unit tests on VBE using rubber duck. I just want to trigger the tests va Jenkins…

   Rubberduck VBA *December 22, 2017* *Reply*

   Thing is, that implies having an Office install on a server, which isn’t a recommended setup. Then your code must be written in such a way that it’s impossible to bring up any kind of dialog (from the VBA code or the host application, or RD, or another add-in), otherwise you have to way to know your server is stalled and waiting for a non-existing user to click some OK button. But to trigger the tests you would need RD to have some kind of API that can invoke the test runner in the add-in instance that’s loaded in the VBE… which is quite a complex thing to do. I don’t think it’s possible ATM. In VB6 a test project would be its own executable so that could possibly work, but in VBA… not convinced. Having a test runner for VBA code is already a feat =)

3. **cornking03** *July 12, 2018* *Reply*

   I’m still new to unit testing and I’m trying to follow your example by implementing your code. Right now, both of the sample tests provided in the article are failing, so I’m trying to figure out where I went wrong. Is there enough code in your example to successfully run the unit tests or was the code in this article more for demonstrating the concepts? I might be getting hung up on all the different classes implemented – in the conclusion, you mention that there are 5 classes, but I seem to count 7: MyTestableMacro, IWorksheetService, IDataService, MyDbDataService, Sheet1Service, MyDataServiceStub, MyWorksheetServiceStub.

   Rubberduck VBA *July 12, 2018* *Reply*

   I wasn’t counting the MyTestableMacro, MyDbDataService and Sheet1Service classes as overhead; the “live” code needs these classes too! 😃
   What’s not working exactly? Are you getting an error? I need to upload an actual downloadable example workbook..

   1. **cornking03** *July 12, 2018*

      I imagine I’m just getting mixed up as the code from each class evolves in the article.
      A downloadable example workbook would be great! =)

4. **SmileyFtW** *November 6, 2018* *Reply*

If I understand correctly, a user form should completely own the information as supplied to it for presentation. If there are filtered aspects to that data it would seem that there would necessarily be logic in the code behind. How would that be tested? For instance suppose there is a combo box that can display a full list of selections or a couple of subsets of the full list. Each subset is selectable by the user via option buttons. If one of the subsets has no members then the ability to choose that option should be disabled.

Rubberduck VBA *November 6, 2018* *Reply*

In such complex cases it's definitely helpful to extract a class that holds the "model", i.e. pull it out of the form itself, and into its own separate, dedicated class. Similar to WPF/MVVM's "ViewModel", you can have an "AvailableThings" property which is your current (filtered) list of available…things; with the filtering logic implemented outside of the form and merely assigning collection properties, it's now easy to test whether your filtering logic works. The option buttons just wire-up to this "ViewModel" by handling their respective "Change" event and invoking the appropriate filtering methods on the "ViewModel" class; the class can raise its own events, that the view can handle – e.g. changing the "AvailableThings" property could raise a "AvailableThingsChanged" event that the view could handle in some private "model_AvailableThingsChanged" procedure, which would then be responsible for updating the actual list of available things. This is a bit hard to describe in a little comment box, I hope I'm making sense 🙂