# Lab Module 1

## Introduction

In automated testing it is common to use objects that look and behave like their production equivalents but are actually simplified. This reduces complexity, allows to verify code independently from the rest of the system and sometimes it is even necessary to execute self-validating tests at all. A Test Double is a generic term used for these objects.

We are going to explore the 3 different types of unit tests (value, stated & interaction based) as well as what are the different types of test doubles before installing a Mocking framework which may lead developer to misunderstandings, influence test design and increase test fragility and refactoring.

## Pre-requisites

Visual Studio 2015, 2017, 2019 and GitHub account.

## Objectives

After completing this lab, you will be able to:

* Add a unit test project to your solution and snippets
* Design your own Fakes, Stubs and Mocks
* Understand the Strategy Pattern.

Scenario

We are writing a program for a smart home microcontroller, and one of the requirements is to automatically turn on the all the devices that are online.

Another requirement is being able to perform firmware updates on offline devices

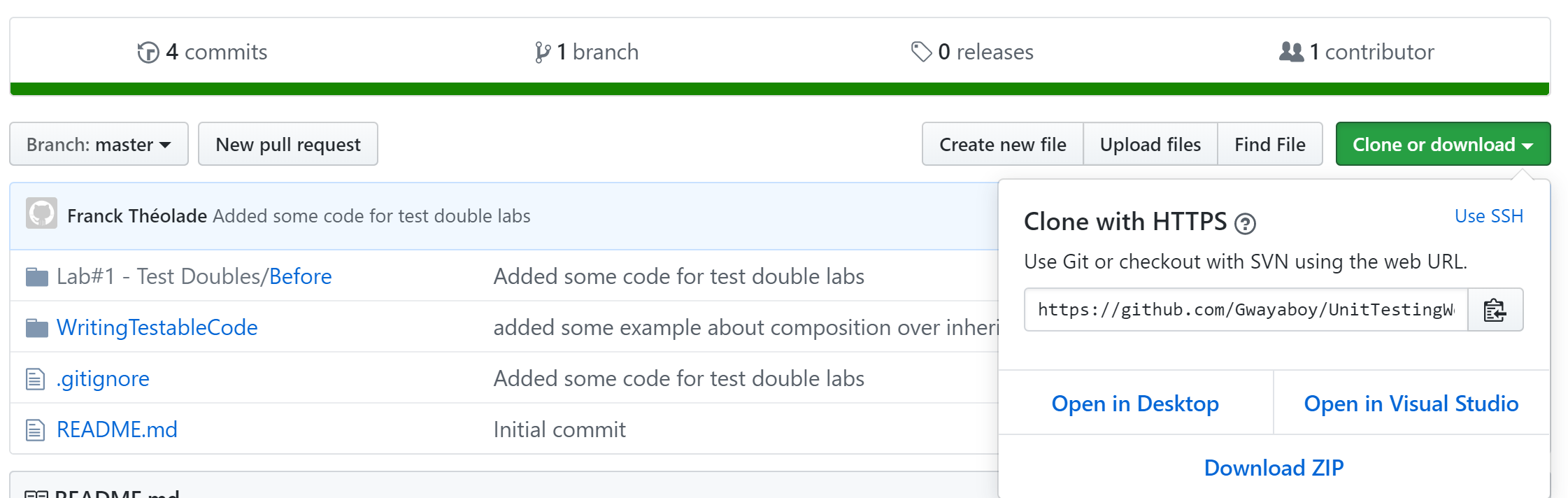
Lastly, we’ll look into implementing a scheduled firmware update on online devices as well.

## Setting up

Download and clone the repository

git clone <https://github.com/Gwayaboy/UnitTestingWorkshop>

Or clone navigate to the URL above and choose clone with Visual Studio:



### Exercise 1: Write first value-based unit test with stub

#### Introduction

1. This code Snippet will ensure consistent test naming convention
2. Add value-based unit

#### Tasks 1.1: Install unit test code snippet

1. Download and save the snippet below

<https://gist.githubusercontent.com/osmyn/906c917653a30864cb52dee02c36c14e/raw/68ec9bd89ea2686ca2adeb540ef2759a07e7bbfd/unittest.snippet>

into your visual studio’s code snippet directory.

For example, if you are using VS2019:

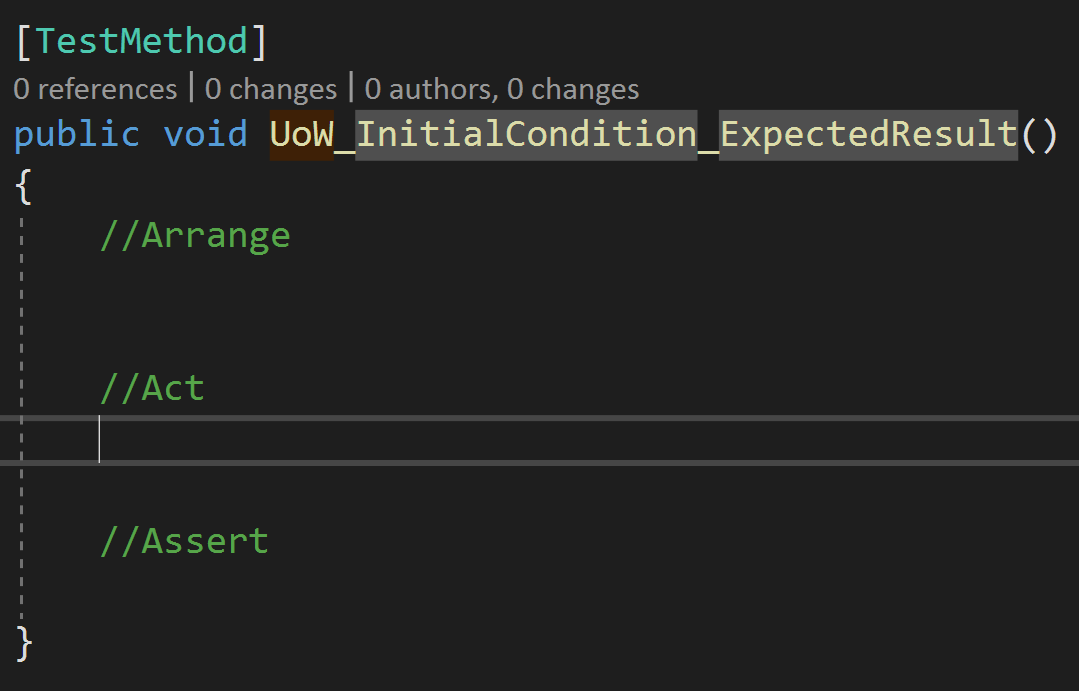
*%USERPROFILE%\Documents\Visual Studio 2019\Code Snippets\Visual C#\My Code Snippets\*

1. Name your snippet something like unittest.snippet.
2. In Visual Studio, you will then just type **ut** and hit tab twice to use this snippet.
3. If you have ReSharper installed, you will either need to switch back to Visual Studio Intellisense in ReSharper’s options or use Ctrl+K, X to bring up the snippet menu.

#### Task 1.2: Add a state-based unit test around turning on all online devices in the controller

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as whether the device is turned on.

1. Add a Unit Test Project by right clicking to the TestDouble solution and choose from contextual menu add -> new project
2. Choose MS Test Project (.Net Core) and name the project UnitTestProject
3. Delete and add a new DeviceControllerTest class to that project and using the “**ut”** code snippet to scaffold your test method as below:



1. Give your test method a meaningful name like TurOnDevices\_FromAllDevices\_OnlyOnlineShouldBeTurnedOn
2. Create a (nested) FakeDeviceRepository implementing IDeviceRepository that will expose a property to hold an in-memory list of Devices:

private class FakeDeviceRepository : IDeviceRepository

{

public IEnumerable<Device> AllDevices { get; set; }

}

1. Next introduce a stub Device engine that will keep track on whether the device got switched on/off by overriding the TurnOn method.
2. Now utilise the FakeRepository and Stub Devices to setup everything required in your Arrange block to create an instance of your DeviceController (SUT). Apart from the FakeRepository you can ignore all other dependencies

var sut = new DeviceController(fakeDeviceRepository, null,null);

Note: Please come up with a few offline devices in your collection for that test and subscribe to the many principle (3 minimum items to cover start, middle and end)

1. Write an assertion that verify all online devices are actually turned on and the offline one remained turned off.

#### Task 1.2: Add a value-based unit test that verifies GetTimeOfDay returns correct portion of day based on passed date time.

1. Add a new DeviceTest class to the unit test project and add a method using a test method that verifies that when passed 6am, the method should return morning

[TestMethod]

public void GetTimeOfDay\_For6AM\_ReturnsMorning()

{

// Arrange phase is empty: testing static method, nothing to initialize

var sut = new Device(Guid.NewGuid(),"MyTV",null);

// Act

var result = sut.GetTimeOfDay(new DateTime(2019, 08, 06, 06, 00, 00));

// Assert

Assert.AreEqual(TimeOfDay.Morning, result);

}

1. Value based unit tests are perfect candidate to be driven by data, let’s extend this test to verify all scenario using MSTest’s [DataRow] attribute.

Let’s rename the test method name above and change its signature to accept hour and expected time of day:

[TestMethod]

[DataRow(0,TimeOfDay.Night)]

[DataRow(6,TimeOfDay.Morning)]

public void GetTimeOfDay\_ForDateTime\_ReturnsTimeOfDay(int h,TimeOfDay expected)

### Exercise 2: Writing and running your first Test

#### Introduction

1. Write your first Pester unit tests that verifies the “MyEborTestModule” module exists.
2. Write a test execution script to run your test.

#### Objectives

After completing this exercise, you will be able to:

1. Understand Pester’s Domain Specific Language (DSL) and its AAA (Arrange, Act and Assert) structure.
2. Write focused and self-explanatory readable unit tests.

#### Tasks 1.1: Writing your first Pester unit test

Following the Behaviour-Driven Development (BDD) pattern, a Pester test file AAA (Arrange-Act-Assert) uses the keywords [**Describe**](https://github.com/pester/Pester/wiki/Describe), [**It**](https://github.com/pester/Pester/wiki/It) and [**Should**](https://github.com/pester/Pester/wiki/Should).

These are nothing more than PowerShell functions accepting parameters and nesting each other in the enumerated order above (if you are curious you can find these function source code on [Pester’s GitHub repository](https://github.com/pester/Pester/tree/master/Functions))

**Describe** and **It** both accepts a mandatory *-Name* parameter to respectively specify a description of the logical group of tests and the expected test outcome

**Should** accepts a mandatory -*Operator* parameter and is used within an **It** blocks to perform an assertion by comparing objects or throwing test failures when test expectations fail. All assertion operators can be negated with the -Not parameter

Write your first test!

1. Write a test that verifies the “MyEborTestModule” module by asserting both the root module “MyEborTestModule.psm1” and the manifest “MyEborTestModule.psd1” files exist under the “MyEborTestModule” module folder. Use the [*-Exist*](https://github.com/pester/Pester/wiki/Should#exist) **Should** operator

$here = Split-Path -Parent $MyInvocation.MyCommand.Path

Describe 'MyEborTest Module Tests' {

Context 'Module Setup' {

It "has the root module MyEborTestModule.psm1" {

"$here\MyEborTestModule\MyEborTestModule.psm1" | Should -Exist

}

}

}

1. Within your Context block after the It block above, write another assertion to further verify module is properly set-up within the same Test file to verify the manifest file contains the name of module using the [*-FileContentMatch*](https://github.com/pester/Pester/wiki/Should#filecontentmatch) **Should** operator

It "has the manifest file of MyEborTestModule.psd1" {

"$here\MyEborTestModule\MyEborTestModule.psd1" | Should -Exist

}

1. (Bonus) After the It block above, write another assertion to check the module contains valid PowerShell

It "Is valid PowerShell (Has no script errors)" {

$contents = Get-Content -Path "$here\MyEborTestModule.psm1" -ErrorAction SilentlyContinue

$errors = $null

$null = [System.Management.Automation.PSParser]::Tokenize($contents, [ref]$errors)

$errors.Count | Should Be 0

}

#### Tasks 1.2: Executing your pester unit test

From Pester version 3.0, any \*.Tests.ps1 script file can be run directly. It could be useful when quickly wanting to execute a test. However, by doing so, you will only see test output at the console and lose any of the benefits of using Invoke-Pester, such as:

* NUnit XML files or output objects with **-OutputFormat** NUnitXML
* Code Coverage analysis with **-CodeCoverage**
* Exit code from PowerShell.exe with **-EnableExit**
  + Which will cause Invoke-Pester to exit with a exit code equal to the number of failed tests once all tests have been run. Use this to "fail" a build when any tests fail.
* Filtering of tests to execute, and a status summary of tests executed / failed when the run is complete with **-Tag** and **-TestName**

More details on parameters at <https://github.com/pester/Pester/wiki/Invoke-Pester>

Write your test runner script!

1. Write a PowerShell script named “run-tests.ps1” in the same directory as your test file using Invoke-Pester to just run the test you just completed

$here = (Split-Path -Parent $MyInvocation.MyCommand.Path)

Set-Location $here

Import-Module Pester

#Could run all tests in all files named with \*.Tests.ps1

Invoke-Pester "$here\MyEborTestModule.Tests.ps1"

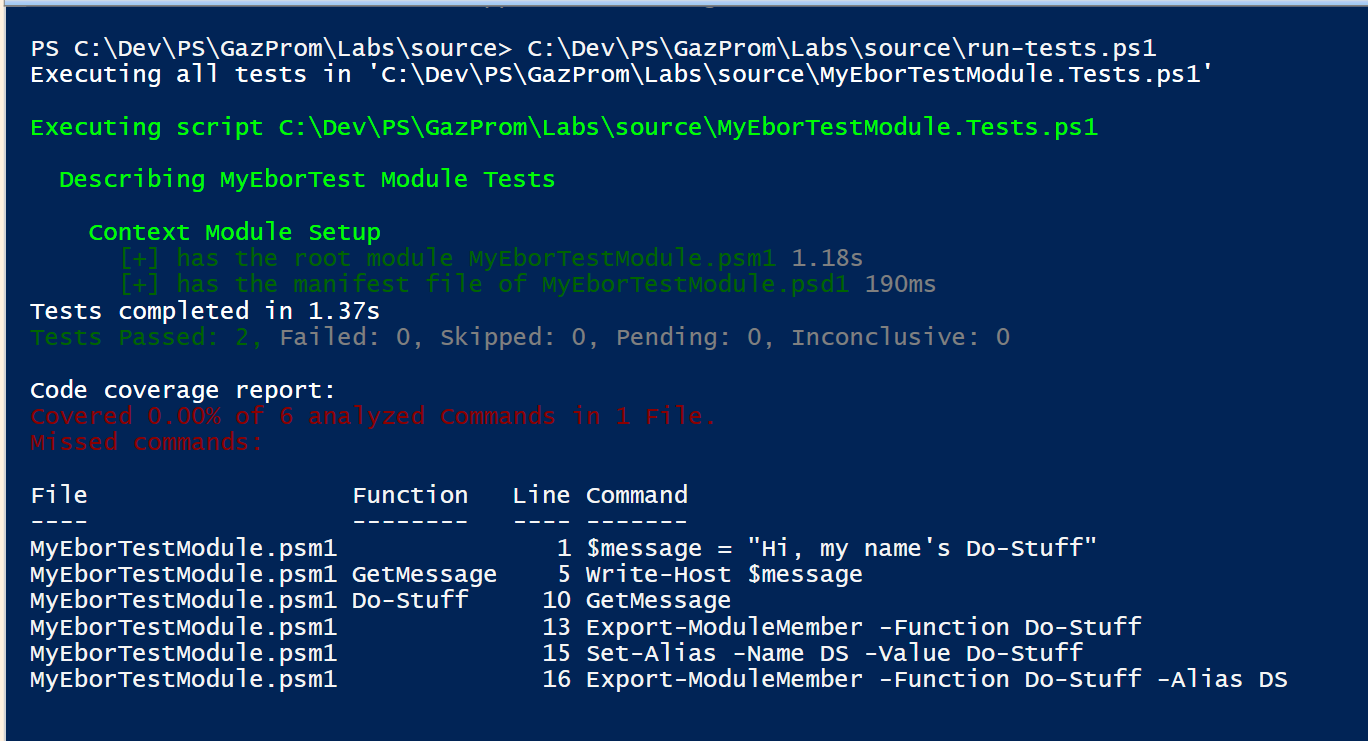
"$here\MyEborTestModule\MyEborTestModule.psm1"

1. Add Code Coverage targeting to your Pester test script (*MyEborTestModule.Tests.ps1*)

Invoke-Pester "$here\MyEborTestModule.Tests.ps1" -CodeCoverage

"$here\MyEborTestModule\MyEborTestModule.psm1"

After executing your tests with coverage your command prompt should look as below:



#### Tasks 1.3: Isolating file operations with TestDrive

Write a test that verifies the function below can append some footer text to an existing text file.

function Add-Footer($path, $footer) {

Add-Content $path -Value $footer

}

1. Create a new AddFooter.Tests.ps1 file and copy the function above
2. Since the function is defined within the test file, we don’t need to import module or load any function in scope.

We are also going to add the scaffold or our test as follows and introduced a Context block to demonstrate how TestDrive controls isolation

Describe "Adds Footer to text file" {

Context "Adding simple footer Footer to text file" {

It "Should add a simple footer" {

}

}

Context "Different execution context" {

It "No drive is defined " {

}

}

}

1. Let’s now arrange by creating a temporary file with some pre-defined content in our first context

Context "Adding simple footer Footer to text file" {

$testpath = "TestDrive:\testDocument.txt"

Set-Content $testPath –Value "Some content."

It "Should add a simple footer" { … }

}

1. Next, we can now invoke our SUT (System Under Test) which is our Add-Footer function

Context "Adding simple footer Footer to text file" {

#Arrange by creating a temporary txt file with some content with TestDrive

$testpath = "TestDrive:\testDocument.txt"

Set-Content $testPath –Value "Some content."

#Invokes the function Add-Footer under test

Add-Footer $testPath "--- My Pretty Footer ----"

$result = Get-Content $testPath

It "Should add a simple footer" { … }

}

1. Finally let’s verify the text file has the pre-existing content appended with our footer content:

It "Should add a simple footer" {

(-join $result) | Should Be "Some content.--- My Pretty Footer ----"

}

The [join](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_join?view=powershell-6) operator concatenates a set of strings into a single string. The strings are appended to the resulting string in the order that they appear in the command.

1. To get the full path, you can use this snippet

function GetFullPath{ Param([string] $Path)

return $Path.Replace('TestDrive:', (Get-PSDrive TestDrive).Root)

}