ICT3101 Lab 2 part 1

The main focus of this laboratory is to continue our exploration of Unit Testing and to apply this into the framework of Behaviour Driven Development (BDD). BDD follows the same principles of TDD but with the addition of User Stories and scenarios into the testing process.

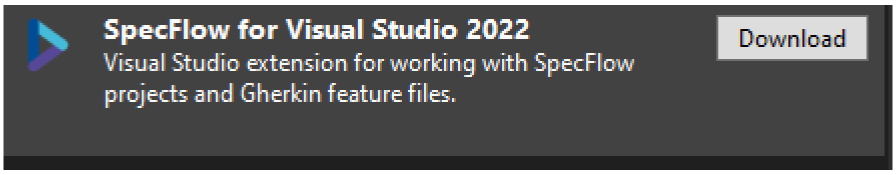
We'll continue with the .Net Core project from the previous lab: "ICT3101_Calculator". Like the previous one, there will be a series of instructions interwoven with the theory of the concepts we are tackling. This first part will be more of a tutorial introduction with part 2 involving more involved tasks.

[SpecFlow on Mac: There are a few good options: 1) use the Git repo: https://github.com/straighteight/SpecFlow-VS-Mac-Integration to integrate with VS MAC. 2) use Rider IDE (Rider: The Cross-Platform .NET IDE from JetBrains), the 30-day free trial will be enough for all the labs. Or 3), use a virtual machine with Windows etc. Rider won't be exactly the same, but you should be able to complete the labs without significant issues.]

Steps:

Setup

1. Get the following VS 2022 Extensions:



SpecFlow for Visual Studio 2022
Visual Studio extension for working with SpecFlow projects and Gherkin feature files.

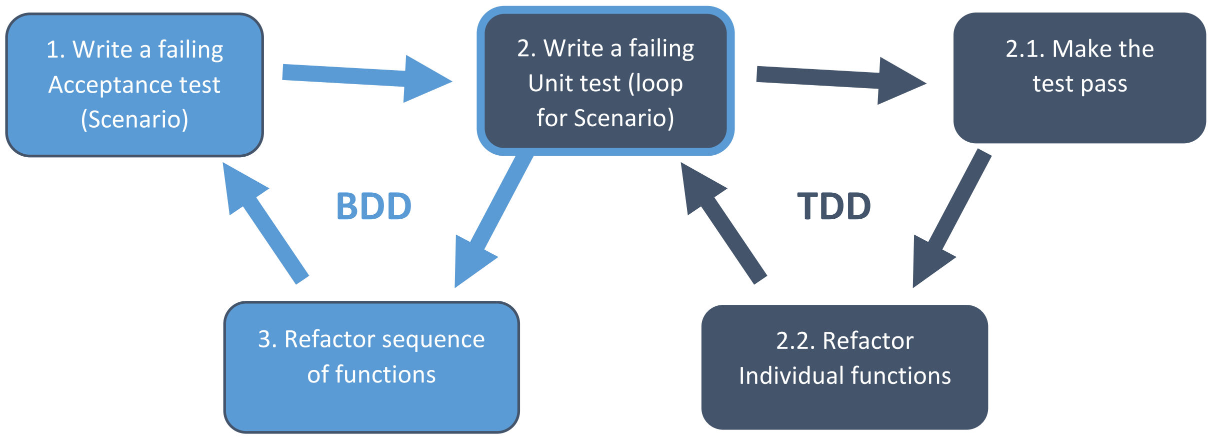
2. We're also going to setup a GitHub repo. This used to be an Extension, but now it's just part of the base VS2022. This can be accessed via Git Changes/Team Explorer tabs on your Solution. But… this will be for later.

Behaviour Driven Development (BDD) Fundamentals:

Behaviour Driven Development (BDD) is a software development process that originally emerged from Test Driven Development (TDD). According to Dan North, who is responsible for the evolution of BDD, "BDD is using examples at multiple levels to create a shared understanding and surface uncertainty to deliver software that matters." The goal for BDD is to use example scenarios to illustrate the behaviour of the system and have them written in a readable and understandable language—I.e., for everyone involved in the development.

One of the main issues with TDD is that tests are still written by developers, without involvement from the client. This is fine for verifying the specification of a function or even a class, but not of the system itself. Furthermore, the knowledge required to write Unit tests is normally outwith the vernacular of the client.

The concept of capturing the clients' requirements, in the form of a system specification, being directly translated into tests and verified brought about BDD. In BDD the client proposes Features which are broken down into sets Use Cases in the form of Scenarios and then translated into the simple language of Gherkin (given, when, then). From here a scenario is mapped to a sequence of Unit tests which can be checked in an automatable fashion. The cycle of BDD involves the TDD cycle, as illustrated below.


1. Write a failing Acceptance test (Scenario)
2. Write a failing Unit test (loop for Scenario)
2.1. Make the test pass
BDD
TDD
3. Refactor sequence of functions
2.2. Refactor Individual functions

BDD with SpecFlow

*We define a **Feature** as follows:*

Feature: UsingCalculatorAdditions

In order to avoid mistakes

As a calculator user

I want to be told the sum of two numbers

And a **Scenario** as follows (Gherkin language):

Scenario: Add two numbers

Given I have entered 50 into the calculator

And I have also entered 70 into the calculator

When I press add

Then the result should be 120 in the screen

BDD Principles

BDD is focussed on providing a shared process and shared tools promoting communication to the software developers, business analysts and stakeholders to collaborate on software development, with the aim of delivering software products with business value. Specifically, BDD considers what a system should do and not on how it should be implemented. Ideally, BDD provides better readability and verifies not only that the software is working but also that it meets the customer's expectations.

Adding the Behaviour

Now let's add a SpecFlow project to our solution.

3. Right click on your solution and select add new project. Name it SpecFlowCalculatorTests
    a. Ensure it's using **.NET 6.0** (SpecFlow isn't yet supported for .NET 8.0), and NUnit as the Test Framework. It will now populate some folders for the Features and Step definitions (that we'll be creating in the rest of this lab). There should be a default feature file and step definitions file. These files are bound to one another. We will start by editing the feature file to have some more useful test scenarios.

4. Now, we're ready to start creating features. By adding in the Feature definitions, we're making the file more human-readable. While adding the Scenarios we're defining the *acceptance criteria* for our program.

   Rename Calculator feature to "UsingCalculatorAddition", and Replace the existing Scenario with the following:

```
@Addition
Scenario: Add two numbers
	Given I have a calculator
	When I have entered 50 and 70 into the calculator and press add
	Then the result should be 120
```

[Note that in the numbers 50, 70 and 120 will be parameterized. This makes our test steps more flexible in future.]

Now it's almost time to generate our test steps—one possible issue to resolve first.

5. Next, ensure that you add the project dependencies to this project (the same as what you did for the Unit Tests project).

6. Now select "CalculatorStepDefinitions" and replace the existing code with the following. (Note that there is a named step for each of the Gherkin notations: "Given", "When" and "Then".)

```
using ICT3101_Calculator;
using NUnit.Framework;

namespace SpecFlowCalculatorTests.StepDefinitions
{
    [Binding]
    public sealed class UsingCalculatorStepDefinitions
    {
        private Calculator _calculator;
        private double _result;

        [Given(@"I have a calculator")]
        public void GivenIHaveACalculator()
        {
            _calculator = new Calculator();
        }
        [When(@"I have entered (.*) and (.*) into the calculator and press add")]
        public void WhenIHaveEnteredAndIntoTheCalculator(double p0, double p1)
        {
            _result = _calculator.Add(p0, p1);
        }

        [Then(@"the result should be (.*)")]
        public void ThenTheResultShouldBeOnTheScreen(int p0)
        {
            Assert.That(_result, Is.EqualTo(p0));
        }
    }
}
```

7. After this you should be able to test the scenario. You can right-click on the scenario then run the tests. Additionally, you ought to be able to see it in the Test Explorer. Make sure all tests pass. (If the tests are failing, the messages in the Test Explorer should help identify the issue.)

8. Okay, so assuming that everything is now working… Another feature of SpecFlow, similar to the NUnit tests we wrote previously, is the use of Scenario Outlines. The following, inexplicable, user scenario is provided in the SpecFlow Gherkin example below. Add this scenario into your Feature file using the "Addition" tag, implement the required steps and then edit the Add function so that the new tests pass (they should all fail to start with: use the test results and error messages to resolve.).

```
@Addition
Scenario Outline: Add zeros for special cases
        Given I have a calculator
        When I have entered <value1> and <value2> into the calculator and press add
        Then the result should be <value3>
        Examples:
        |value1 |value2 |value3 |
        |1      |11     |7      |
        |10     |11     |11     |
        |11     |11     |15     |
```

9. Since you better understand features and scenarios, let's create another new feature file, as follows:

```
Feature: UsingCalculatorDivision
        In order to conquer divisions
        As a division enthusiast
        I want to understand a variety of division operations

@Divisions
Scenario: Dividing two numbers
        Given I have a calculator
        When I have entered 1 and 2 into the calculator and press divide
        Then the division result should be 0.5

@Divisions
Scenario: Dividing zero by a number
        Given I have a calculator
        When I have entered 0 and 15 into the calculator and press divide
        Then the division result should be 0

@Divisions
Scenario: Dividing by zeros
        Given I have a calculator
        When I have entered 15 and 0 into the calculator and press divide
        Then the division result equals positive_infinity

@Divisions
Scenario: Dividing by zero by zero
        Given I have a calculator
        When I have entered 0 and 0 into the calculator and press divide
        Then the division result should be 1
```

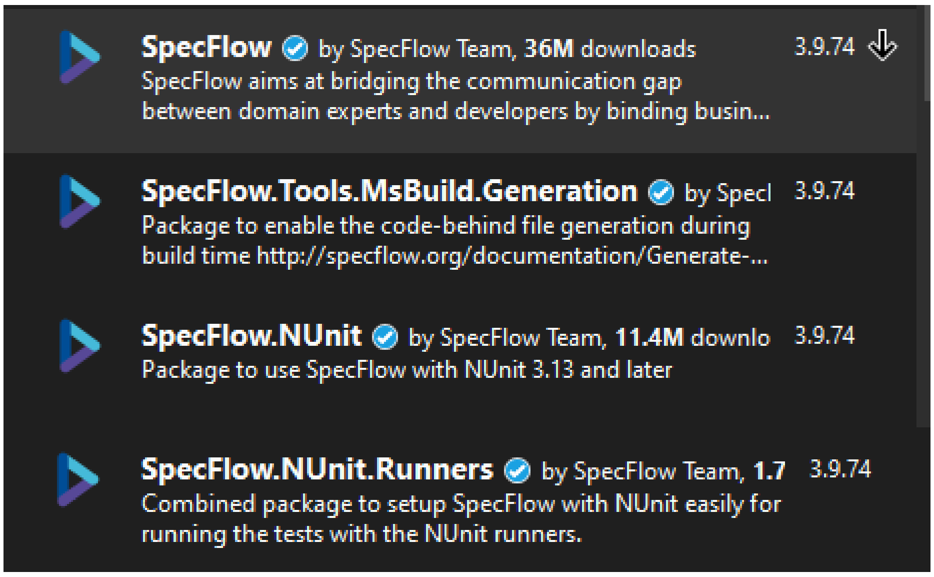
10. Now set up the associated step definitions class for these scenarios (you can right-click on a step definition and click "Define Steps" to help autogenerate a template for you).

    Note that you'll have to make further edits to your current calculator code in order to pass all these tests—possibly adjusting previous tests too. However, there're issues to solve before that in the Division steps class… see below.

11. SpecFlow reuses common step definitions for Gherkin statements in its Features. In our example this includes "**Given** I have a calculator"; hence, we have two separate step definition classes bound to our new scenario. This will cause you an issue to run, why? See if you can resolve this with a SpecFlow technique called "Context Injection".

—The end of part 1 questions—

EDIT, all of the relevant packages should now be automatically installed. However, the following Nuget packages will be needed for the SpecFlowProject.

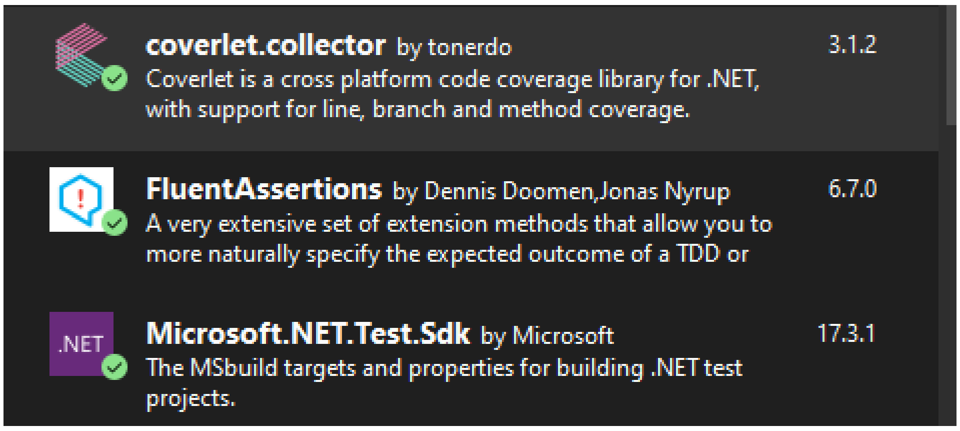


SpecFlow by SpecFlow Team, 36M downloads 3.9.74
SpecFlow aims at bridging the communication gap between domain experts and developers by binding busin...

SpecFlow.Tools.MsBuild.Generation by Specl 3.9.74
Package to enable the code-behind file generation during build time http://specflow.org/documentation/Generate-...

SpecFlow.NUnit by SpecFlow Team, 11.4M downlo 3.9.74
Package to use SpecFlow with NUnit 3.13 and later

SpecFlow.NUnit.Runners by SpecFlow Team, 1.7 3.9.74
Combined package to setup SpecFlow with NUnit easily for running the tests with the NUnit runners.



coverlet.collector by tonerdo 3.1.2
Coverlet is a cross platform code coverage library for .NET, with support for line, branch and method coverage.

FluentAssertions by Dennis Doomen,Jonas Nyrup 6.7.0
A very extensive set of extension methods that allow you to more naturally specify the expected outcome of a TDD or

Microsoft.NET.Test.Sdk by Microsoft 17.3.1
The MSbuild targets and properties for building .NET test projects.