**Behavior Driven Development (BDD)**

**Cucumber/Specflow**

# Introduction

[BDD](#_Behavior_Driven_Development) Day 01

[Features](#_Features) Day 01

[BDD Tools Cucumber & SpecFlow](#_BDD_Tools_Cucumber) Day 01

[Why BDD Framework?](#_Why_BDD_Framework?) Day 01

# Hands-On

[Getting Started](#_Getting_Started) Day 02

[Setting Up your SpecFlow Project](#_Setting_Up_your) Day 02

[Adding a Feature File](#_Adding_a_Feature) Day 02

[Generating Step Definitions](#_Generating_Step_Definitions) Day 02

[Executing Your First Test](#_Executing_Your_First) Day 02

[Adding a Calculator Class](#_Adding_a_Calculator) Day 02

[Referencing the Calculator Class](#_Referencing_the_Calculator) Day 02

[Binding the First Given Statement](#_Binding_the_First) Day 02

[Binding the Second Given Statement](#_Binding_the_Second) Day 02

[Binding the When Statement](#_Binding_the_When) Day 02

[Binding the Then Statement](#_Binding_the_Then) Day 02

[Final CalculatorSteps.cs Code](#_Final_CalculatorSteps.cs_Code) Day 02

[Final Calculator.cs Code](#_Final_Calculator.cs_Code) Day 02

[Feature Files – Coding Convention](#_Feature_Files_–) Day 02

[Step Definitions – Coding Convention](#_Step_Definitions_–) Day 02

[Reports](#_Reports) Day 02

[More Information](#_More_Information) Day 02

[References](#_References) Day 02

## Behavior Driven Development

BDD is an iterative development process. Each iteration starts with a set of tests written for a new piece of functionality.  We write tests first and then add application code.

## Features

* Tests are written in plain descriptive English type grammar
* Tests are explained as behavior of application and are more user focused
* Collaboration between Business stakeholders, Business Analysts, QA Team and developers
* Consumes natural language that non-technical stakeholders can understand
* Driven by Business Value
* BDD frameworks such as Cucumber or JBehave are an enabler, acting a “bridge” between Business & Technical Language

BDD is popular and can be utilized for Unit level test cases and for UI level test cases. Tools like RSpec (for Ruby) or in .NET something like MSpec or SpecUnit is popular for Unit Testing following BDD approach.  Alternatively, you can write BDD-style specifications about UI interactions. Assuming you’re building a web application, you’ll probably use a browser automation library like WatiR/WatiN or Selenium, and script it either using one of the frameworks I just mentioned, or a given/when/then tool such as Cucumber (for Ruby) or SpecFlow (for .NET).

## BDD Tools Cucumber & SpecFlow

### Cucumber

**Cucumber** is a testing framework which supports **Behavior Driven Development (BDD).**It lets us define application behavior in plain meaningful English text using a simple grammar defined by a language called **Gherkin**. Cucumber itself is written in **Ruby**, but it can be used to “test” code written in ***Ruby*** or other languages including but not limited to ***Java***, ***C#*** and ***Python.***

### SpecFlow

**SpecFlow**is inspired by ***Cucumber*** framework in the Ruby on Rails world. ***Cucumber*** uses plain English in the Gherkin format to express user stories. Once the user stories and their expectations are written, the Cucumber gem is used to execute those stores. **SpecFlow brings the same concept to the .NET world**and allows the developer to express the feature in plain English language. It also allows writing specification in human readable[Gherkin format](http://toolsqa.wpengine.com/cucumber/gherkin-business-driven-development-bdd-language/).

## Why BDD Framework?

Let’s assume there is a requirement from a client for an E-Commerce website to increase the sales of the product with implementing some new features on the website. The only challenge of the development team is to convert the client idea in to something that actually delivers the benefits to client.

The original idea is awesome. But the only challenge here is that the person who is developing the idea is not the same person who has this idea. If the person who has the idea happens to be a talented software developer, then we might be in luck: the idea could be turned into working software without ever needing to be explained to anyone else. Now the idea needs to be communicated and has to travel from Business Owners (Client) to the development teams or many other people.

Most software projects involve teams of several people working collaboratively together, so high-quality communication is critical to their success. As you probably know, good communication isn’t just about eloquently describing your ideas to others; you also need to solicit feedback to ensure you’ve been understood correctly. This is why agile software teams have learned to work in small increments, using the software that’s built incrementally as the feedback that says to the stakeholders “Is this what you mean?”

Below image is the example of what clients have in their mind and communicated to the team of developers and how developers understands it and work on it.

With the help of Gherkin language cucumber helps facilitate the discovery and use of a ubiquitous language within the team. Tests written in cucumber directly interact with the development code, but the tests are written in a language that is quite easy to understand by the business stakeholders. Cucumber test removes many misunderstandings long before they create any ambiguities in to the code.

## Getting Started

## Installation and Setup

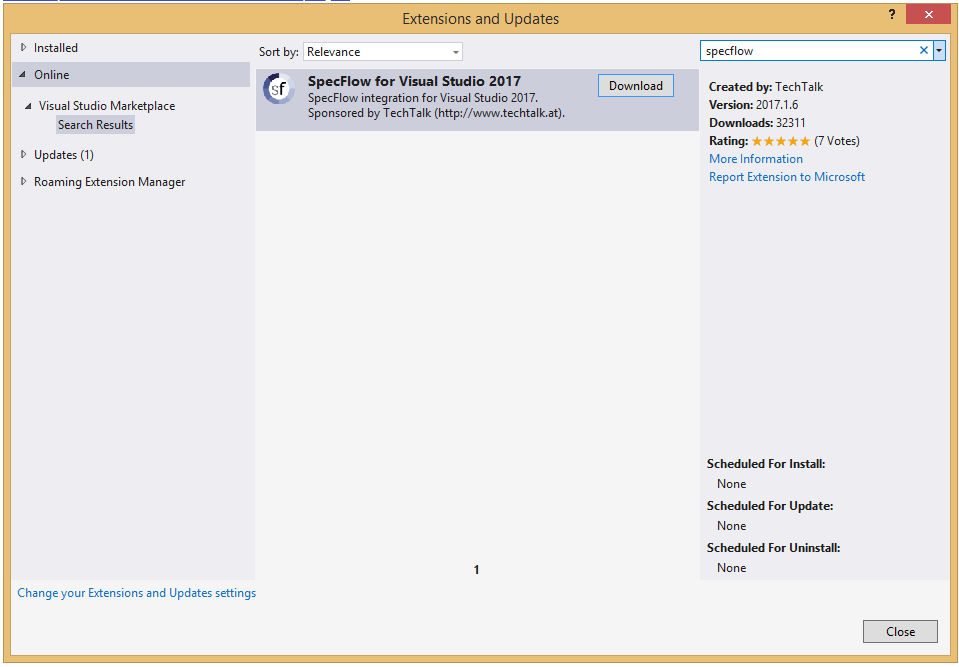
Installing SpecFlow consists of two steps:

1. Install the IDE integration
2. Set up your Visual Studio project to work with SpecFlow

## Installing the IDE Integration Packages

The method used to install the IDE Integration packages depends on your IDE:

* Visual Studio 2010+ (including Community Edition but not Express Edition): The easiest method is to select Tools | Extensions and Updates from the menu in Visual Studio, switch to the Online search on the left and enter “SpecFlow” in the search field at the top right.



## Setting Up your SpecFlow Project

To set up your specification project:

* 1. Add “Unit Test Project” to your solution (e.g. “MyProject.Specs”).  
     Note: Creating a “Unit Test Project” is the recommended procedure, as it reduces the number of steps required to set up your project.
  2. You can choose to remove the UnitTestX.cs file, as it is not required.
  3. Add SpecFlow+ Runner to your specification project using NuGet:
     1. Right-click on your solution (e.g. “MyProject”) and select Manage NuGet Packages for Solution.
     2. Search for “SpecRun” and install SpecRun.SpecFlow.Alternatively, you can install the package from NuGet’s console (Tools | NuGet Package Manager | Package Manager Console) as follows:  
        PM> Install-Package SpecRun.SpecFlow

Installing SpecFlow+ Runner automatically downloads the SpecFlow runtime from NuGet and adds it to your specifications project.

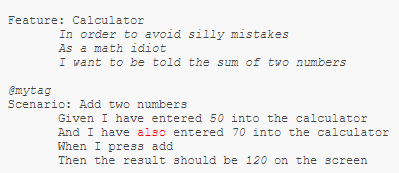
**Note**: Instead of SpecFlow+ Runner, you can also use [other test engines](http://specflow.org/documentation/Unit-Test-Providers/), like MsTest, xUnit, MbUnit or NUnit. These packages are installed in exactly the same manner as SpecFlow+ Runner. However to follow all the steps in this guide, you need to install SpecFlow+ Runner. The evaluation version of SpecFlow+ Runner delays the execution of your tests by a number of seconds each time.

## Adding a Feature File

You now need to add a feature file to your specifications project that outlines a specific feature and includes a test scenario:

* 1. Right-click on your specifications project and select **Add | New Item** from the popup menu.
  2. Select **SpecFlow Feature File** (restrict the entries by entering “SpecFlow” in the search field), give it a meaningful name and click on **Add**.

The feature file is added to your specification project. It includes a default scenario for adding two numbers. You can leave most of the content as is, but add the word “also” to the scenario on the line beginning with “And” (see red text below):



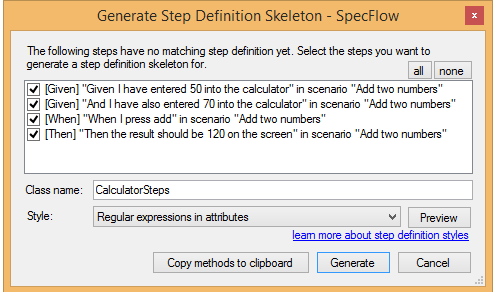
We will use this scenario to demonstrate the first development iteration. By adding “also” to the second Given statement, we have two separate statements that will be parsed differently (otherwise both statements would match the regular expression “I have entered (.\*) into the calculator”). This both simplifies the code in our example and illustrates how different statements linked with “And” are handled.

You can also assign tags to scenarios (e.g. “@mytag” in the example above), which can be used to filter scenarios, and control how scenarios are executed and automated. For more details on using tags, see [Hooks](http://specflow.org/documentation/Hooks/), [Scoped Bindings](http://specflow.org/documentation/Scoped-Bindings/), [FeatureContext](http://specflow.org/documentation/FeatureContext/) and [ScenarioContext](http://specflow.org/documentation/ScenarioContext/) in the documentation.

## Generating Step Definitions

In order to test our scenario, we need to create step definitions that bind the statements in the test scenario to the application code. SpecFlow can automatically generate a skeleton for the automation code that you can then extend as necessary:

1. Right-click on your feature file in the code editor and select Generate Step Definitions from the popup menu. A dialogue is displayed.
2. Enter a name for the class, e.g. “CalculatorSteps”.



Click on Generate and save the file. A new skeleton class is added to your project with steps for each of the steps in the scenario:



## Executing Your First Test

SpecFlow+ Runner integrates with Visual Studio Test Explorer. After adding your first specification and building the solution, the business readable scenario titles will show up in Visual Studio Test Explorer:

1. Build your solution.
2. Select Test | Windows | Test Explorer to open the Test Explorer:

Scenarios are displayed with their plain text scenario title instead of a generated unit test name.

## Click on Run All to run your test. As the automation and application code has not yet been implemented, the test will not pass successfully.

## Implementing the Automation and Application Code

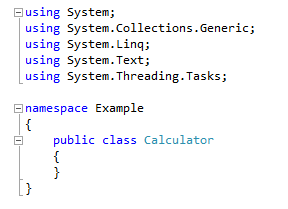
In order for your tests to pass, you need to implement both the application code (the code in your application you are testing) and the automation code (binding the test scenario to the automation interface). This involves the following steps, which are covered in this section:

1. Reference the assembly or project containing the interface you want to bind the automation to (including APIs, controllers, UI automation tools etc.).
2. Extend the step definition skeleton with the automation code.
3. Implement the missing application code.
4. Verify that the scenario passes the test.

## Adding a Calculator Class

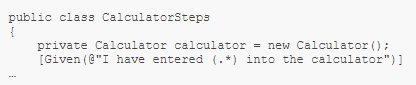
The application code that implements the actual functions performed by the calculator should be defined in a separate project from your specification project. This project should include a class for the calculator and expose methods for initialising the calculator and performing the addition:

1. Right-click on your solution in the Solution Explorer and select **Add | Project** from the context menu. Choose to add a new class library and give your project a name (e.g. “Example”).
2. Right-click on the .cs file in the new project and rename it (e.g. “Calculator.cs”), and choose to rename all references.
3. Your new class should be similar to the following:



## Referencing the Calculator Class

1. Expand your specification project and right-click on References. Select Add Reference from the context menu.
2. Click on Solution on the left of the Reference Manager dialogue. The projects in your solution are listed.
3. Enable the check box next to the Example project to reference it from the specifications project.
4. Click on OK.  
   A reference to the Example project is added to the References node in the Solution Explorer.
5. Add a using directive for the namespace (e.g. “Example”) of your Calculator class to the CalculatorSteps.cs file in your specification project:  
   using Example;
6. Define a variable of the type Calculator in the CalculatorSteps class prior to the step definitions:



Defining a variable outside of the individual steps allows the variable to be accessed by each of the individual steps and ensures the variable is persistent between steps.

## Implementing the Code

Now that the step definitions can reference the **Calculator** class, you need to extend the step definitions and implement the application code.

## Binding the First Given Statement

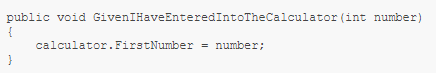
The first Given statement in the scenario needs to initialise the calculator with the first of the two numbers defined in the scenario (50). To implement the code:

1. Open CalculatorSteps.cs if it is not already open.  
   The value defined in the scenario is passed as a parameter in the automation code’s associated function, e.g.:

InitialGivenStep

Rename this parameter to something more human-readable (e.g. “number”):  
public void GivenIHaveEnteredIntoTheCalculator(int number)

1. To initialise the calculator with this number, replace ScenarioContext.Current.Pending(); in the step definition as follows:



1. Switch to the file containing your **Calculator** class (e.g. Calculator.cs) and add a public integer member to the class:

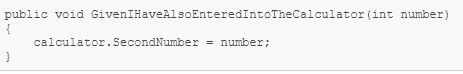
public int FirstNumber { get; set; }

You have now determined that the FirstNumber member of the **Calculator** class is initialized with the value defined in the scenario when the test is executed.

#### Binding the Second Given Statement

The second Given statement in the scenario needs to initialize the second number with the second value defined in the scenario (70). To implement the code:

1. Open CalculatorSteps.cs if it is not already open.
2. Locate the function corresponding to the second Given statement and rename the p0 parameter to “number”, as before.
3. To initialize the calculator with the second number, replace ScenarioContext.Current.Pending(); in the step definition as follows:



1. Switch to the file containing your Calculator class and add another public integer member to the class:

public int SecondNumber { get; set; }

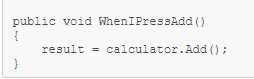
You have now determined that the SecondNumber member of the Calculator class is initialised with the value defined in the scenario when the test is executed

## Binding the When Statement

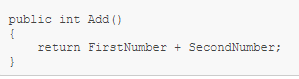
The step for the When statement needs to call the method that performs the actual addition and store the result. This result needs to be available to the other final step in the automation code in order to verify that the result is the expected result defined in the test scenario.

To implement the code:

1. Open CalculatorSteps.cs if it is not already open.
2. Define a variable to store the result at the start of the CalculatorSeps class (before any of the steps):  
   private int result;  
   Defining a variable outside of the individual steps allows the variable to be accessed by each of the individual steps.
3. Locate the function corresponding to the When statement and edit it as follows:



1. Switch to the file containing your Calculator class and define the Add() method:

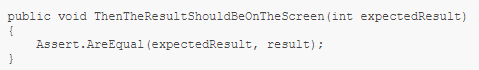


You have now determined that the Add() method of the calculator class is called once the initial Given steps have been performed

## Binding the Then Statement

The step for the Then statement needs to verify that the result returned by the **Add()** method in the previous step is the same as the expected result defined in the test scenario. To implement the code:

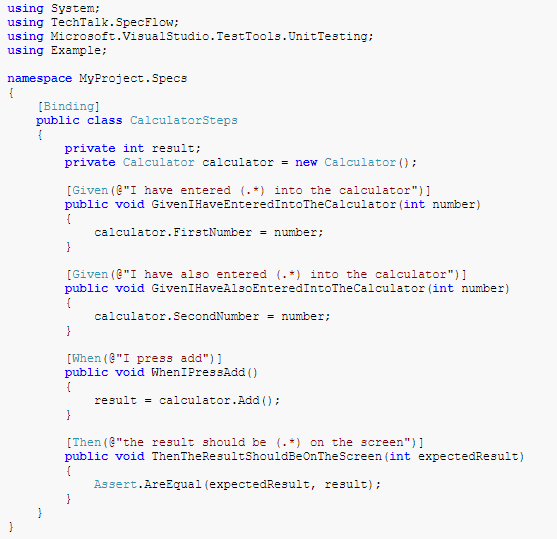
1. Open CalculatorSteps.cs if it is not already open.  
   As the result will be verified using Assert, you need to add “using Microsoft.VisualStudio.TestTools.UnitTesting;” to the top of your automation code.
2. Locate the function corresponding to the Then statement. Rename the p0 parameter in the function call (this time to “expectedResult”) and edit the step definition as follows



You have now implemented the final piece of the jigsaw – testing that the result returned by your application matches the expected result defined in the scenario.

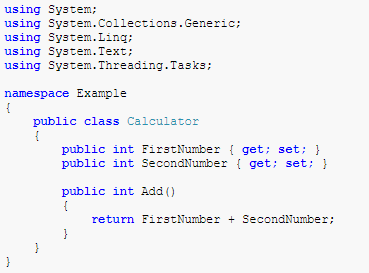
## Final CalculatorSteps.cs Code

Your CalculatorSteps.cs code should be similar to the following:



## Final Calculator.cs Code

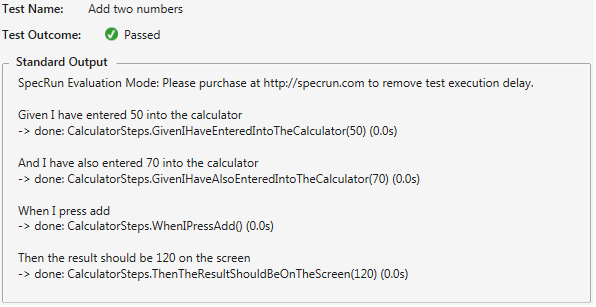
Your Calculator.cs code should be similar to the following:



## Executing the Tests Again

Now that the test steps have been bound to your application code, you need to rebuild your solution and execute the tests again (click on **Run All** in the Test Explorer). You should see that the test now passes (green).

Click on Output in the Test Explorer to display a summary of the test:



## Feature Files – Coding Convention

When you add a feature file to your specifications project, it outlines a specific feature and includes a test scenario:

* Make sure full description is maintained in every feature file for FEATURE keyword.
* Make sure we mention appropriate description for every SCENARIO which describes what test case is actually doing.
* Make sure we use Gherkin keywords correctly.
  + **Given** - The purpose of Givens is to put the system in a known state before the user starts interacting with the system (in the When steps). Avoid talking about user interaction in givens. If you were creating use cases, givens would be your preconditions.

Ex:

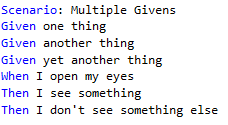
* + - 1. Create records (model instances) / set up the database state.
      2. Login a user
* **When** - The purpose of When steps is to describe the key action the user performs

Ex:

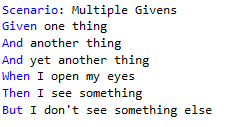
* + - 1. Interact with a web page (Selenium interaction like click, check, etc.).
      2. Interact with some other user interface element.
* **Then** - The purpose of Then steps is to observe outcomes. The observations should be related to the business value/benefit in your feature description. The observations should also be on some kind of output – that is something that comes out of the system (report, user interface and message) and not something that is deeply buried inside it (that has no business value).

Ex:

* + - 1. Verify that something related to the Given+When is (or is not) in the output
      2. Check that some external system has received the expected message
* **And**, **But** – These should be used in conjunction to your Given, When, Then keywords. For Example - You can use a scenario like this -



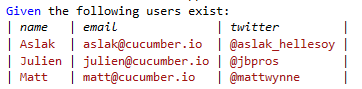
Or you can make it read more fluently by writing like below –



* **Data Tables**: This should be used when we want to pass the Data to a Single Step.

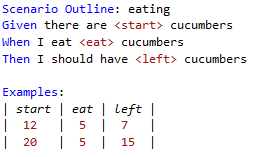
**Note**: Data Tables will be passed to the [Step Definition](https://cucumber.io/docs/reference#step-definitions) as the last argument. The type of this argument will be Data Table.

Ex:



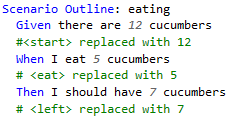
* **Scenario Outline**: This should be used when we want to pass the Data to the entire Scenario. It allow us to more concisely express these examples through the use of a template with placeholders, using Scenario Outline, Examples with tables and < > delimited parameters:

Ex:



A Scenario Outline is run once for each row in the Examples section beneath it (not counting the first row). The way this works is via placeholders <dataToBeReplace>.

The scenario that is actually run is:



* Note: It will repeat for remaining rows.

## Step Definitions – Coding Convention

Supported Step Definitions Attributes are

[Given(regex)] or [Given]

[When(regex)] or [When]

[Then(regex)] or [Then]

[StepDefinition(regex)] or [StepDefinition] - matches for given, when or then attributes

* + DO NOT duplicate step definition methods. Instead annotate a single method with multiple attributes in order to support different phrasings in the feature file for the same automation logic. For Example –

[When(@"I perform a simple search on '(.\*)'")]

[When(@"I search for '(.\*)'")]

public void WhenIPerformASimpleSearchOn(string searchTerm)

{

...

}

**Step Definition Method Rules:**

* + Must be in a public class, marked with the [Binding] attribute.
  + Must be a public method.
  + Can be either a static or an instance method. If it is an instance method, the containing class will be instantiated once for every scenario.
  + Cannot have out (or) ref parameters.
  + Cannot have a return type.

**Step Definition Method Names:**

* + Method names should be concatenated with Underscores (or) should be in Pascal Case

Ex:

With Underscores:

[Given] public void Given\_I\_have\_entered\_NUMBER\_into\_the\_calculator (int number)

{ ... }

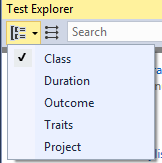
With Pascal Case:

[Given] public void GivenIHaveEnteredNUMBERIntoTheCalculator(int number) {... }

## Calling Steps from Step Definitions

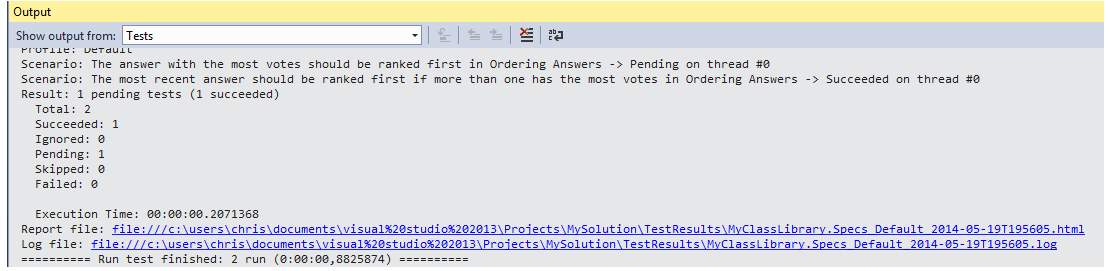
Even though we have a feature to call a step from another step in Step Definition file, Please DO NOT USE this feature. As our automation suite keeps on increase with more number of steps, if someone changes the step description, then we may end up in failing the test cases as we need to update in all the references in which the step is used.

## Grouping Tests by Category



## Reports

SpecFlow can generate reports once your tests have finished executing that includes a breakdown of the results of your tests. The default report includes a statistical overview of the status of all tests, as well as information on individual scenarios, including Gherkin test cases, statistics on the total number and percentage of successful tests, and the execution time for each step. When running tests from within Visual Studio, a link to the generated report(s) is included in the **Output** window once the tests have completed.



Click on the link to the report file to view the report in Visual Studio:



## More Information

* Class Names, Method Names, WebElement names should be in Pascal case. Ex: AMHomePage, SelectAccessLevel, SubmitButton, PasswordTextField etc.
* Method Arguments name, Local variable names should be in Camel case. Ex:

userNameText, textFromPage etc.,

* Make sure alignment is maintained.
* Remove any unwanted spacing between lines.
* Make sure class names ends with word ‘Page’, to identify which page it refers to. Ex: AccessManagerLoginPage, AMLoginPage, AMHomePage, AMRequestPage etc.
* Make sure class name and the corresponding file names are same.
* Please use the below Folder Structure for easy navigation and look and feel.
  + FeatureFiles
    - ProjectName1
      * FeatureFile1.feature
      * FeatureFile2.feature
    - ProjectName2
      * FeatureFile1.feature
      * FeatureFile2.feature
  + StepDefinitions
    - ProjectName1
      * ProjectNameStepDef.cs
    - ProjectName2
      * ProjectNameStepDef.cs
  + Any Re-usableFolder
    - Hooks.cs
    - Any Helper files which are reused in the framework. Like DB, API, API Classes, Excel(Data Driven) etc.,
  + Remaining files like App.config, Default.srprofile remains same.
* You can have multiple Feature files, but maintain only one Step Definition file for one project.

Ex: For AccessManager project, please maintain 1 step definition file AMStepDef.cs

For CSS project, 1 step def CSSStepDef.cs file.

* DO NOT change App.config, Default.srprofile unless you know what it is going to effect. Contact Automation Point of Contact before you make any changes to those files.
* Unless it is privacy related, make use of Data Table (or) Scenario Outline for passing the data to the scenario. Else make use of Excel to store privacy data.
* Make sure you close all objects if you are using any Excel Application for Data Driven.
* Make sure you close the DB connection, REST API connections etc. at the end of the each scenario, if you are opening a connection before any scenario. Use [BeforeScenario] and [AfterScenario] attributes in hooks file to achieve this.
* Make sure every scenario is independent of each other, because dependent scenarios will become brittle and fail at any point of the execution.

## References

<http://specflow.org/getting-started/>

<https://github.com/techtalk/SpecFlow.Plus.Examples>

<https://specflow.org/plus/documentation/Reports/>

<https://specflow.org/plus/documentation/SpecFlowPlus-Runner-Profiles/#report/>

<https://specflow.org/plus/documentation/Tutorial:-Customising-Reports/>

<https://github.com/techtalk/SpecFlowPlus-Resources/wiki/SpecFlowPlus-Runner-Profiles#settings->

<https://github.com/techtalk/SpecFlow.Plus.Examples>