

Why SpecFlow+ Excel?

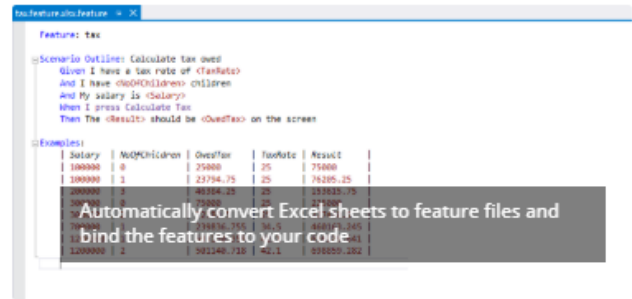
SpecFlow+ Excel is currently not compatible with SpecFlow 3 and .NET Core. It is only compatible with SpecFlow 2.4 or earlier.

Spreadsheets are commonly used to describe rules and calculations in the financial and engineering sectors. Excel is convenient for working with large amounts of tabular data, using formulas to perform calculations (equivalent to abstract specifications or rules) and displaying results (equivalent to example sets). This approach is well suited to [Specification by Example](#), BDD and of course SpecFlow!

SpecFlow+ Excel allows you to define requirements and example sets in Excel. Your requirements can be used in the same way as plain text Gherkin files. You can:

- Define entire features in Excel using the worksheets as scenarios
- Extend scenario outline examples in normal plain text feature files with examples in Excel tables

SpecFlow+ Excel seamlessly integrates into the SpecFlow development tool chain. However you can also access it from the command line, meaning you can use it with any tool in the Cucumber family, such as Cucumber, Cucumber-JVM or Behat. The command line tool works on non-Windows platforms using Mono.



Buy SpecFlow+
€ 159 / user

Request Trial

SpecFlow + licenses are perpetual and include 1 year of support and upgrades to new versions. Click [here](#) for more details on licenses.

Got a sales question? [Contact us!](#)

1. Prepare Feature Files For External Examples

Reference : - <https://specflow.org/plus/documentation/Prepare-Feature-Files-for-External-Examples/>

```
@source:excel-file-path[:sheet-name]
```

Feature: Calculator

Scenario Outline: Add two numbers

Given I have entered <a> into the calculator
And I have entered into the calculator
When I press add
Then the result should be <result> on the screen

```
@source:CalculatorExamples.xlsx:sheet1
```

Examples:

```
| case | a | b | result |
```

	A	B	C	D	E
1	case	a	b	result	
2	classic	50	70	120	
3	simple	2	3	5	
4					

2. Define entire features in Excel using the worksheets as scenarios

Reference :- <https://specflow.org/plus/excel/getting-started/>

Installing the SpecFlow+ Excel and SpecFlow+ Runner Packages

To install SpecFlow+ Excel

1. Create a new solution (e.g. Calculator.sln) in Visual Studio.
2. Add a second project (e.g. "MySpecs") to your solution as your specifications project. You can remove the .cs file from this second project, as it is not required.
3. Add SpecFlow+ Excel and SpecFlow+ Runner¹ to your specification project using NuGet:
 1. Right-click on your solution and select Manage NuGet Packages for Solution.
 2. Search for "SpecFlow.Plus.Excel" and install the package SpecFlow+ Excel package. Alternatively, you can install the package from NuGet's console (Tools | NuGet Package Manager | Package Manager Console) as follows:

```
PM> Install-Package SpecFlow.Plus.Excel -ProjectName
```
 3. Search for "SpecRun.Runner" and install the SpecFlow+ Runner package. Alternatively, you can install the package from NuGet's console (Tools | NuGet Package Manager | Package Manager Console) as follows:

```
PM> Install-Package SpecRun.Runner -ProjectName
```

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

¹. Instead of SpecFlow+ Runner, you can also use [other test engines](#), like MsTest, xUnit, MbUnit or NUnit. These packages are installed in exactly the same manner as SpecFlow+ Runner.

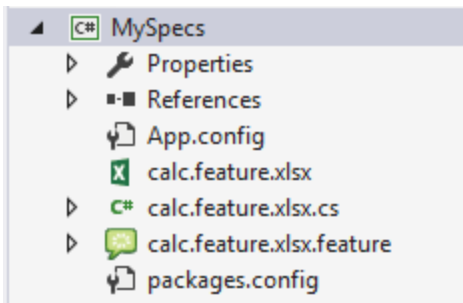
Your First Excel Test

The next step is to create an Excel file that contains our test scenario. This test scenario will test that the calculator correctly adds two numbers together. This test scenario will be identical to the test scenario in the [SpecFlow+ Runner Getting Started tutorial](#), but in Excel format.

1. Create a new, empty Excel spreadsheet and save it with the .feature.xlsx file extension (e.g. Calculator.feature.xlsx) in your specification project's folder. This file extension is used by SpecFlow+ Excel to identify Excel files containing your tests.
2. Rename the first sheet to "Add two numbers". This will be the name of the scenario and is displayed in the Test Explorer in Visual Studio.
3. Define your test on the first sheet as follows:

	A	B	C	D
1	Given	I have entered	50	into the calculator
2	And	I have also entered	70	into the calculator
3	When	I press add		
4	Then	the result should be	120	on the screen

4. Save your Excel file.
5. Return to Visual Studio and add the Excel file to your specifications project.
6. Build your project. 2 files are generated in the project directory (where your Excel file is located): Calculator.feature.xlsx.cs and Calculator.feature.xlsx.feature. Add these files to your specifications project as well.



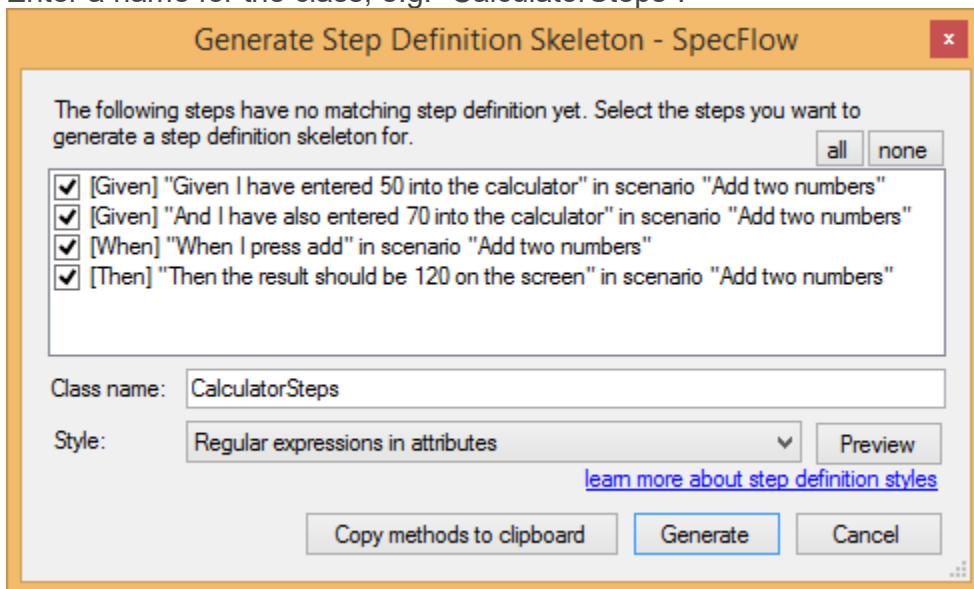
You can optionally mark the generated files as dependent on the Excel file, so that they are displayed as a sub-item in Visual Studio.

7. Click on the .feature.xlsx.feature file you just added, and remove “SpecFlowSingleFileGenerator” from the **Custom Tool** field in the file’s properties.
8. Rebuild your project. A test should be added to the test explorer with the same name as the worksheet.

Binding Your Test

You now need to bind your test to your application code. The first step is to generate the step definitions:

1. Switch to the feature file (calc.feature.xlsx.feature) in Visual Studio. You will see the test you defined in Excel formatted as plain text.
2. Right-click on your feature file in the code editor and select **Generate Step Definitions** from the popup menu. A dialogue is displayed.
3. Enter a name for the class, e.g. “CalculatorSteps”.



4. 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:

```
using System;
using TechTalk.SpecFlow;

namespace MyProject.Specs
{
    [Binding]
    public class CalculatorSteps
    {
        [Given(@"I have entered (.*) into the calculator")]
        public void GivenIHaveEnteredIntoTheCalculator(int p0)
        {
            ScenarioContext.Current.Pending();
        }

        [Given(@"I have also entered (.*) into the calculator")]
        public void GivenIHaveAlsoEnteredIntoTheCalculator(int p0)
        {
            ScenarioContext.Current.Pending();
        }

        [When(@"I press add")]
        public void WhenIPressAdd()
        {
            ScenarioContext.Current.Pending();
        }

        [Then(@"the result should be (.*) on the screen")]
        public void ThenTheResultShouldBeOnTheScreen(int p0)
        {
            ScenarioContext.Current.Pending();
        }
    }
}
```

Executing Your First Test

Build your application and run your test from the Test Explorer. At the moment, the test will not pass because the necessary application code has not been implemented yet.

Note: The evaluation version of SpecFlow+ Excel generates an extra scenario in each Excel feature file with title "SpecFlow+ Excel Evaluation Mode". This scenario has a single step, which is unbound by default, so this test fails as "pending" when executed. You can [request a demo license key](#) to remove this restriction.

Implementing the Automation and Application Code

Note: If you have already completed the [Getting Started guide for SpecFlow+ Runner](#), you can reuse the code from that project, as this section is identical. In this case you can skip to the [next section](#). You can also proceed to the end of this section and copy the source code directly from there.

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. Either rename the default project in your solution, or, if necessary, 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 project and rename it (e.g. "Calculator.cs"), and choose to rename all references.
3. Your new class should be similar to the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Example
{
    public class Calculator
    {
    }
}
```

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:

```
public class CalculatorSteps
{
    private Calculator calculator = new Calculator();

    [Given(@"I have entered (.*) into the calculator")]
    ...
}
```

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.:

```
[Given(@"I have entered (.*) into the calculator")]
public void GivenIHaveEnteredIntoTheCalculator(int p0)
```

2. Rename this parameter to something more human-readable (e.g. "number"):

```
public void GivenIHaveEnteredIntoTheCalculator(int number)
```

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

```
public void GivenIHaveEnteredIntoTheCalculator(int number)
```

```
{
```

```
    calculator.FirstNumber = number;
```

```
}
```

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

```
public int FirstNumber { set; private get; }
```

You have now determined that the FirstNumber member of the **Calculator** class is initialised 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 initialise 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 initialise the calculator with the second number, replace `ScenarioContext.Current.Pending();` in the step definition as follows:

```
public void GivenIHaveAlsoEnteredIntoTheCalculator(int number)
```

```
{
```

```
    calculatorSecondNumber = number;
```

```
}
```

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

```
public int SecondNumber { set; private get; }
```

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 { get; set; }
```

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:

```
public void WhenIPressAdd()
```

```
{
```

```
    result = calculator.Add();
```

```
}
```

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

```
public int Add()
```

```
{
```

```
    return FirstNumber + SecondNumber;
```

```
}
```

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 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:

```
public void ThenTheResultShouldBeOnTheScreen(int expectedResult)
```

```
{
```

```
    Assert.AreEqual(expectedResult, result);
```

```
}
```

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:

```
using System;
using TechTalk.SpecFlow;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Example;

namespace MyProject.Specs
{
    [Binding]
    public class CalculatorSteps
    {
        private int result { get; set; }

        private Calculator calculator = new Calculator();

        [Given ("I have entered (.*?) into the calculator")]
        public void GivenIHaveEnteredIntoTheCalculator(int number)
        {
            calculator.FirstNumber = number;
        }

        [Given ("I have also entered (.*?) into the calculator")]
        public void GivenIHaveAlsoEnteredIntoTheCalculator(int number)
        {
            calculator.SecondNumber = number;
        }

        [When ("I press add")]
        public void WhenIPressAdd()
        {

```



```
result = calculator.Add();
```

```
}
```

```
[ Then ( @"the result should be (.*?) on the screen" ) ]
```

```
public void ThenTheResultShouldBeOnTheScreen( int expectedResult)
```

```
{
```

```
Assert.AreEqual(expectedResult, result);
```

```
}
```

```
}
```

```
}
```

Final Calculator.cs Code

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

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

```
using System.Threading.Tasks;
```

```
namespace Example
```

```
{
```

```
public class Calculator
```

```
{
```

```
public int FirstNumber { set; private get; }
```

```
public int SecondNumber { set; private get; }
```

```
public int Add()
```

```
{
```

```
return FirstNumber + SecondNumber;
```

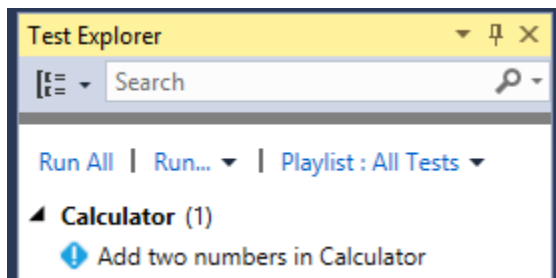
```
}
```

```
}
```

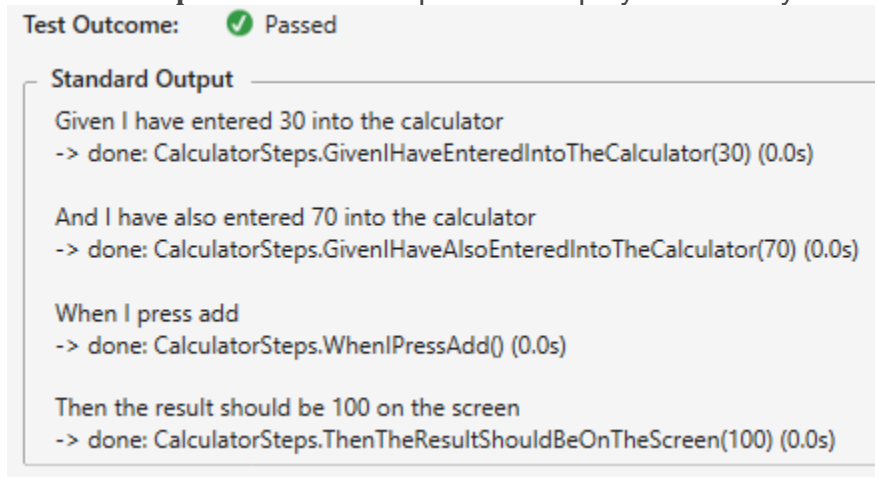
```
}
```

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:



Using Tables for Data

You can define example data in tables in Excel, allowing you to define multiple test cases for a single scenario. We are going to define several sets of data to be used as input for the test case we defined previously (adding two numbers).

1. Open the Excel file again.
2. To specify example data, you need to start a line with “Examples:”. You can also enter a name for the examples in the cell to its right and specify tags on the line above.
3. Move down a row and one column to the left of where you entered “Examples:” to start your table. This indentation is necessary, or you will receive an error and SpecFlow+ Excel will be unable to parse the Excel file. Your table should look like the following:

Then	the result should be	140	on the screen	
Examples:				
	case	a	b	result
	Addition	30	70	100
	Subtraction (negative addition)	25	-10	15
	Adding negative numbers	-5	-1	-6

The first row of the table is a header. The entries in the “case” column are used as the names for each test case. Here we have a test case for two positive numbers, a test case that involves adding a negative number, and a test case adding two negative numbers. You might want to also define a test case that checks that adding zero returns the right result as well.

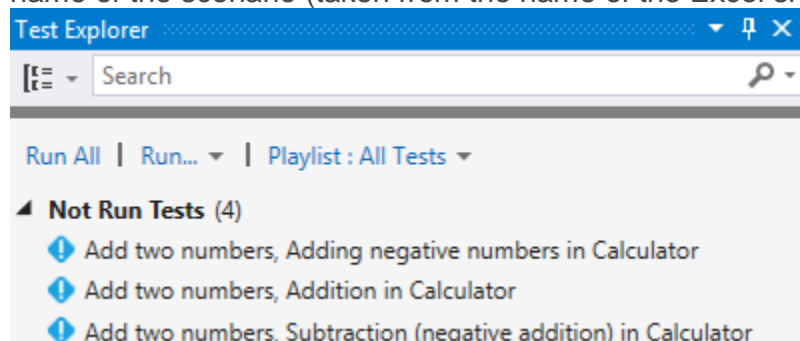
The “a” and “b” columns contain the two numbers serving as input for the calculator, and the result column contains the expected result (the sum of the two numbers in the “a” and “b” columns).

4. We still need to tell SpecFlow+ Excel to use the data in this table in our scenario. Replace the numbers in the Gherkin scenario with the names of the corresponding column in the table, enclosed in angled brackets (< and >) to indicate that they are placeholders. The table should now look like this:

	A	B	C	D	E
1	Given	I have entered	<a>	into the calculator	
2	And	I have also entered		into the calculator	
3	When	I press add			
4	Then	the result should be	<result>	on the screen	
5	Examples:				
6		case	a	b	result
7		Addition	30	70	100
8		Subtraction (negative addition)	25	-10	15
9		Adding negative numbers	-5	-1	-6

When executing the scenario, SpecFlow+ Excel will replace the placeholders with the corresponding values from the table.

5. Save the Excel file.
6. Rebuild your solution in Visual Studio. You should see 3 tests in the test explorer now, including the name of the scenario (taken from the name of the Excel sheet) and the name of the test cases:



7. Run the tests to verify your code is working properly!
8. Open the feature file to view the updated scenario. It should be similar to the following:

```

Scenario Outline: Add two numbers
    Given I have entered <a> into the calculator
    And I have also entered <b> into the calculator
    When I press add
    Then the result should be <result> on the screen

Examples:
    | a | b | result |
    | 30 | 70 | 100 |
    | 25 | -10 | 15 |
    | -5 | -1 | -6 |
  
```

Making the Most of Excel




Until now, we have simply been abusing Excel as a text editor – we could have written our feature file in Notepad to get the same result. The power of using Excel to write your test only begins to show once you start using Excel’s in-built functions to your advantage.

One of the big advantages of Excel is its ability to handle formulas, and you can take advantages of formulas in the Excel sheet. This can be particularly useful in cases where you need to perform complex financial calculations, and where the method of calculating the result may change (e.g. as the result of an

updated tax code). While our formula is going to be simple, it demonstrates how you can use the result of a formula as an input value for your tests.

We will now change the Excel test to be more dynamic.

1. Open your Excel file.
2. Navigate to the first cell containing a numeric value in the “result” column (E7, if your layout is the same as the one used in this guide). Change the contents of this cell to a formula that add the two numbers in the “a” and “b” columns (e.g. “=C7+D7”):

E7	:				<div>=C7+D7</div>
	A	B	C	D	E
1	Given	I have entered	<a>	into the calculator	
2	And	I have also entered		into the calculator	
3	When	I press add			
4	Then	the result should be	<result>	on the screen	
5	Examples:				
6		case	a	b	result
7		Addition	30	70	100
8		Subtraction (negative addition)	25	-10	15
9		Adding negative numbers	-5	-1	-6

3. Select all entries in the “result” column starting from the top, and use fill down (shortcut: Alt+E, I, D) to copy the formula to all cells in the “result” column.
4. You can now change the values in the “a” and “b” columns and the expected result will update accordingly.
5. Save your Excel file and return to Visual Studio.
6. Rebuild your solution and verify that the tests still pass. If you are asked if you want to reload the changed files, answer Yes to ensure that the changes you made are updated in your project’s files.

Converting Between Gherkin and Excel

You can use the SpecFlow+ Excel converter command line tool to convert Excel files to Gherkin syntax and vice versa.

To convert an Excel file to Gherkin:

```
SpecFlow.Plus.Excel.Converter.exe convert EXCEL_FILE_PATH TARGET_FILE_PATH
```

To convert a Gherkin file to Excel:

```
SpecFlow.Plus.Excel.Converter.exe initialize FEATURE_FILE_PATH TARGET_FILE_PATH
```

<p?You can thus convert existing Gherkin scenarios to Excel, rather than needing to redefine existing tests. Note that this is a one-time conversion – you cannot update an existing Excel file. In addition to saving you time converting existing test scenarios, converting Gherkin files to Excel can be a good way to understand how the various test elements are represented in Excel.

More information can be found [here](#).

Helper Formulas and Comments

In some cases you may want to use helper formulas in Excel – intermediate values that are used by calculations, but do not server as input for your test cases. Data entered in a row with at least two empty

columns on the left is ignored by SpecFlow+ Excel:

C	D	E	Ignored by SpecFlow+		G
<a>	into the calculator	Two empty columns		30	
	into the calculator			70	
<result>	on the screen			100	

You can also use “#” to indicate that the contents of a cell should be ignored (e.g. for comments). More information on formatting can be found [here](#).

You may also find it helpful to use Excel’s formatting features (fonts, colours, shading etc.) to distinguish between various elements.