SPECFLOW TUTORIAL

An introduction to using SpecFlow for C sharp in the Visual Studio 2013 environment
Created April, 2015

Edited by

DAVID RICE

TechLink
Bozeman, MT

Contents

1 Overview

This document is intended to provide a very brief and condensed introduction to SpecFlow: the concepts behind it and how to install and implement it in Visual Studio 2013. **If your goal is to simply see SpecFlow work with a simple example, go directly to chapter 4.** The rest of this documentation is kept generic and more conceptually focused.

The concepts of SpecFlow are first addressed by explaining Gherkin – the business readable language that SpecFlow uses. Next the concepts of how and why SpecFlow works are addressed. Finally, the specific steps to install SpecFlow and code examples of how to begin writing tests are shown.

1.1 Domain and Goals

SpecFlow is a specific implementation of the more generic process of Behavior-driven development (BDD) [1] [2].

BDD is a development process which handles the domain of testing. BDD looks to handle the questions of

- Where to start in the testing process
- What to test and what not to test
- How much to test in one go
- What to call the tests
- How to understand why a test fails

SpecFlow is a test-first method that handles automated testing using BDD concepts.

2 Gherkin and SpecFlow Concepts and Syntax

SpecFlow is an environment that uses Gherkin. Thus it is worth understanding what Gherkin is and how it works.

2.1 Concepts

Gherkin is the language that SpecFlow understands. It is a Business Readable, Domain Specific Language (BRDSL) [3] [4] with the intention of describing behavior without explicit need for code implementation.

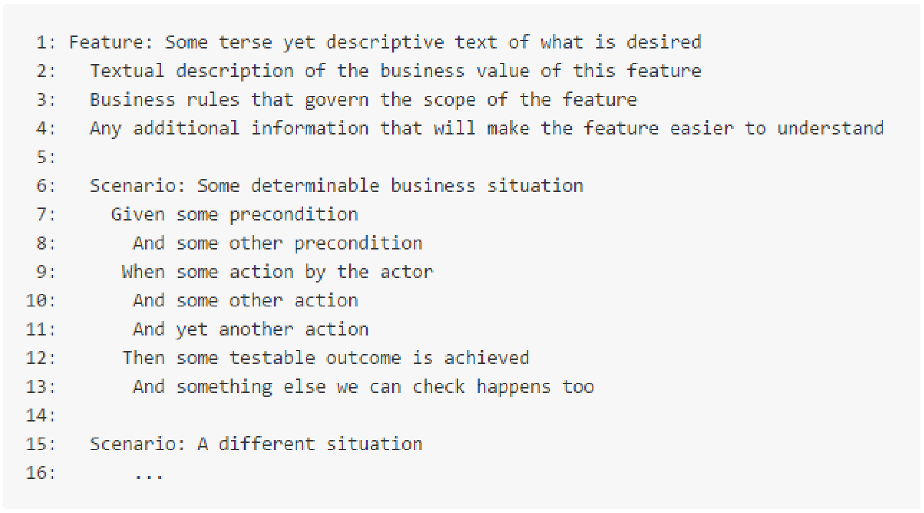
The design of Gherkin provides documentation, automated tests, and a structured way to write code to support the BRDSL.

By convention, each Gherkin source file will have a .feature extension and will contain a description of a single feature with one or more scenarios. One can think of SpecFlow being the software (or more accurately, the environment) while Gherkin is the specific language.

2.2 SpecFlow Syntax

Writing a proper SpecFlow implementation involves writing two files: A .feature file in the Gherkin language and step definition files in the C# language.

2.2.1 The .feature File



```
 1: Feature: Some terse yet descriptive text of what is desired
 2:   Textual description of the business value of this feature
 3:   Business rules that govern the scope of the feature
 4:   Any additional information that will make the feature easier to understand
 5:
 6:   Scenario: Some determinable business situation
 7:     Given some precondition
 8:       And some other precondition
 9:      When some action by the actor
10:       And some other action
11:       And yet another action
12:      Then some testable outcome is achieved
13:       And something else we can check happens too
14:
15:   Scenario: A different situation
16:       ...
```

Figure 1: Generic Gherkin Syntax (borrowed directly from citation [3])

The structural key words of Gherkin are **Feature** and **Scenario**. The logical key words are **Given**, **When**, **Then**, **But**, and **And**.

Lines 2 through 4 in figure 1 are not parsed. Line spacing and indentation is recognized in SpecFlow in a similar fashion to Python. Line endings terminate steps. One feature can contain many scenarios.

2.2.2 Step Definitions

Step Definitions are written in C# (any .NET language) and involve object creation and assertion that the objects, methods, and variables function as they are intended to.

```
using System;
//used for passing variables between step definition methods
//using the ScenarioContext.Current calls
using TechTalk.SpecFlow;
//package that makes asserting more like the English language
using Should.Fluent;

namespace Example.AcceptanceTests.StepDefinitions
{
        [Binding]
        //auto generated from the .feature file
        public class UserSubmitsOnlineExam
        {
                //auto generated from the .feature file
                [Given(@"User is logged in")]
                public void GivenIAmLoggedIn()
                {
                        //begin writing code here
                        var user = new User();
                        ScenarioContext.Current.Set(user);
                        user.LogIn();
                        //below is an assertation that uses
                            the Should.Fluent package
                        user.LoggedInStatus.Should().Be.True();
                }
                [Given(@"I have answered all the questions")]
                public void GivenIHaveAnsweredAllTheQuestions
                {
                        //more code here
                        ...
                }
        }
}
```

Sample code one would see in a step definition

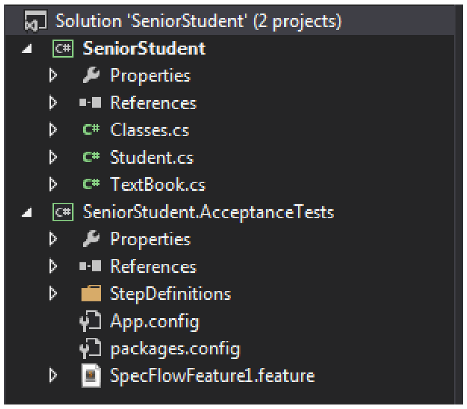
3 Getting SpecFlow up and running in Visual Studio 2013

3.1 Prerequisites

- Using Visual Studio 2013
- The following Extensions and Updates are installed:
  - NuGet Package Manager for Visual Studio 2013
  - NUnit Test Adapter
  - SpecFlow for Visual Studio 2013
- Specflow, Nunit, and ShouldFluent are properly referenced using NuGet. (How to do this is described after step 4 of subsection 4.3).

3.2 How to structure your project

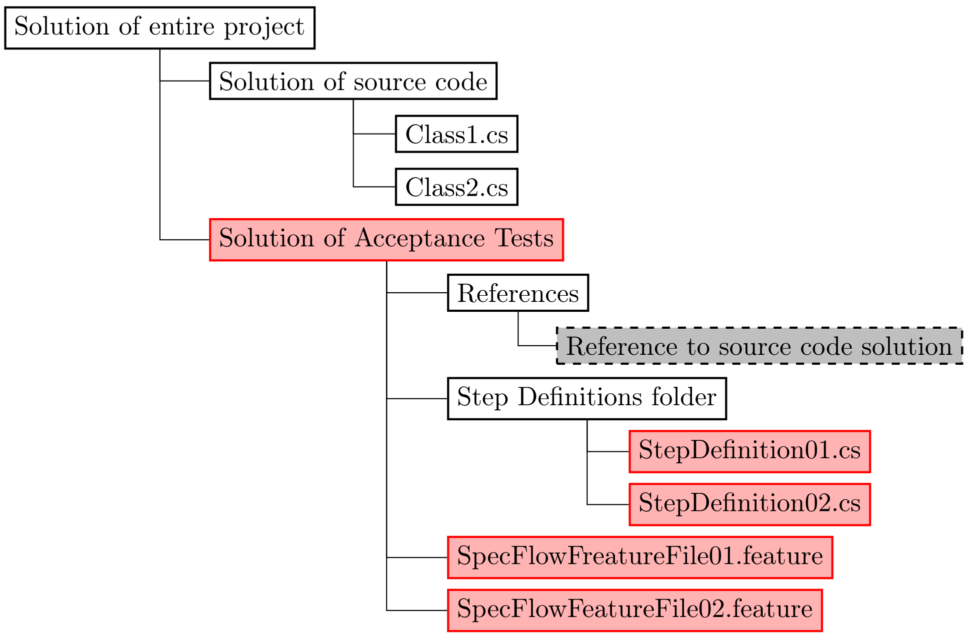
It is best practice to keep all testing classes in a separate Solution (as a reminder, a Solution is Visual Studio's term for general file structure and project management). In this case, the tests will be done in a Class Library Solution (.dll) at the same hierarchy as the Solutions representing the rest of the project (See Figure 2).



```
Solution 'SeniorStudent' (2 projects)
  SeniorStudent
    Properties
    References
    Classes.cs
    Student.cs
    TextBook.cs
  SeniorStudent.AcceptanceTests
    Properties
    References
    StepDefinitions
    App.config
    packages.config
    SpecFlowFeature1.feature
```

Figure 2: Structure

A possible, generic file structure with the key files and Solutions for SpecFlow could look as follows:


Solution of entire project
Solution of source code
Class1.cs
Class2.cs
Solution of Acceptance Tests
References
Reference to source code solution
Step Definitions folder
StepDefinition01.cs
StepDefinition02.cs
SpecFlowFreatureFile01.feature
SpecFlowFeatureFile02.feature

3.3 Standards for creating and naming SpecFlow files and Solutions

3.3.1 The Class Library (.dll) Solution for tests

For the class library *Solution of Acceptance Tests* as seen in the above figure, it is best to name that Solution *Projectname.AcceptanceTests.* For example, if you were working on a project named *WebClientServer*, when you are ready to create the Solution for your tests, made a new class library and name it *WebClientServer.AcceptanceTests.*

Within this Solution it is also necessary to ensure you have the correct references in the *References* section for the three extensions used in a standard SpecFlow testing enviornemnt (TechTalk.Specflow, Should.Fluent, and nunit.framework) as well as a references to your actual project.

To add these references, ensure you are in the acceptance tests Solution and type the following commands into the NuGet package manager (if you do not have NuGet, install it via the extensions manager and open the console with *Tools->NuGet package manager->Package Manager Console)*:

```
// in the following lines, assume the acceptance tests
   solution you are in is named 'Test.AcceptanceTests'
PM> Install-Package SpecFlow -ProjectName Test.AcceptanceTests
PM> Install-Package NUnit -ProjectName Test.AcceptanceTests
PM> Install-Package ShouldFluent -ProjectName
   Test.AcceptanceTests
```

Finally, add the reference to your project and the nunit framework with the following steps:

- Ensure you are within the acceptance tests solution
- Right click *References*
- Click on *Add References*
- In the left side column of the now visible Reference Manager, find and check the following two items (ensure you actually activate the check mark)
  - *Assemblies->Extensions->nunit.framework*
  - *Solution->Projects->ProjectName*

Press *OK* and your project will now have all the proper references to do unit tests with SpecFlow.

3.4 Feature Files

The syntax of feature files is described in Section 2.2.1.

Create a feature file by the following steps:

- While in the Solution Explorer panel, right click your *ProjectName.AcceptanceSteps* Solution
- Click *Add->New item*
- Locate and click *Visual C# items* on the left side column
- Click *SpecFlow Feature File* and give is a sensible name as described below

One can either keep .feature files one level below the acceptance tests solution, or create a file kept one level below the acceptance tests solution named *FeatureFiles.*

A feature file should have a descriptive, camel-case name such as *ClientsLogInToServer.feature.*

3.5 Step Definitions

It is highly recommended to keep step definitions in a personally created folder named *StepDefinitions*.

Generate step definitions by the following steps:

- First, create the folder to hold step definitions
  - Right click *ProjectName.AcceptanceSteps->Add->New Folder*
  - Name it *StepDefinitions*
- Then, use SpecFlow to automatically generate step definitions
  - While in a .feature file
  - Right click anywhere within the code of the feature you want to generate steps for
  - Click on *Generate Step Definitions*
  - Ensure all steps have check marks, click *Generate*
  - Save the file in the *ProjectName/ProjectName.AcceptanceSteps/StepDefinitions* folder you previously created
  - In some situations, you may get an *Inconsistent Line Endings* error. If this error arises, select the *Windows (CR LF)* option and press *Yes*
  - Your task is then to fill in the step definitions to be able to run proper tests
  - Do this for all .feature files

4 Example

4.1 Preface

The following example brings together all the ideas of the previous chapters in a very basic program. A lot of the same code and examples are reused in this example. The redundancy is to help those that want to quickly get started using SpecFlow without reading the previous chapters.

4.2 Code on GitLab

A working Visual Studio project identical to the example about to be shown can be found at
`https://github.com/DavidRiceMontana/SpecFlowFullExample.git`

4.3 Building a project with SpecFlow unit tests from scratch

Unit testing is a powerful and important method. It can even be used during the development and creation of a project. SpecFlow is a Visual Studio extension that attempts to make unit testing during and after development as simple as possible.

SpecFlow uses two files. A feature file and a step definition file. The feature file is code written in the gherkin language that has the file type .feature. For example, a .feature file might look like this:

```
Feature: Student takes an exam
      In order to get through college
  As a student
  I must study and pass exams

Scenario: Prepared Student Takes Exam
  Given I am a student
  And I have read the textbook
  When I take the exam
  Then I will be okay
```

The step definition file is a Class (written in C# in this case) that is automatically generated by the SpecFlow extension leaving you, as the programmer, to fill in the methods in order to eventually get a working unit test.

A step definition file might look like the following figure:

```
using System;
using TechTalk.SpecFlow;
namespace SeniorStudent.AcceptanceTests.StepDefinitions
{
        [Binding]
        public class StudentTakesAnExamSteps
        {
                [Given@("I am a good student")]
                public void GivenIAmAGoodStudent()
                {
                        ScenarioContext.Current.Pending();
                }
        }
}
```

Figure 4.3.1: Auto-generated Step Definition example

It should be obvious then how the step definitions bridge the gap from the English-worded feature files to actual code that can be run as a unit test.

Tests with SpecFlow will use two solutions within the project. One solution is the normal code one would write to make a project while the other solution contains all the code and classes used for testing.

Note: if you are already familiar with how to initially set up a Visual Studio 2013 project, skip equations 1 and 2 and go to the top of the next page (to the paragraph with the hyperlink).

Setting up your project and initial code to test

Open Visual Studio 2013 and create a new project.

$$\text{File -> New -> Project} \tag{1}$$

Create a .dll class library that will be the Solution that holds non-testing code. On the left column go to...

$$\text{Installed -> Templates -> Visual C\#} \tag{2}$$

then select 'Class Library' in the center box.

Name the class library *SeniorStudent* and press *OK*

Before beginning on the test class libraries, it is worth filling in the SeniorStudent Solution with workable code.

The most sensible way to do this is to delete the work just done in equations 1 and 2, go to
`https://github.com/DavidRiceMontana/SpecFlowWithNoTestSolution.git`

which will provide a very basic C# Project called *Senior Student* that will be used to run SpecFlow tests on. Load the code from the above link into Visual Studio.

The testing Solution

Create another .dll class library. This will be the Solution that holds everything relevant to doing tests with SpecFlow. Find the Solution Explorer panel (if not already open in Visual Studio go to *VIEW -> Solution Explorer* to make it visible). Then right click

Solution 'SeniorStudent' (1 project) (3)

And left click

Add -> New Project (4)

Locate *Class Library* which is found in the same location described by Equation 2. After clicking on *Class Library*, name it *SeniorStudent.AcceptanceSteps* then press *OK*.

It is now a good time to do a few behind-the-scenes steps in order to have NUnit (the unit testing extension used in Visual Studio), SpecFlow, and ShouldFluent (a nice package used for logical assertions) as well as having a reference of the *SeniorStudent* Solution to *SeniorStudent.AcceptanceSteps* Solution.
Do the following steps:

- Ensure NuGet is installed in Visual Studio. (If not, install it via the extensions manager)
- Open the NuGet package manager with *Tools -> NuGet Package Manager -> Package manager Console*
- Type the following three commands into the Package Manager Console
  - *Install-Package SpecFlow -ProjectName SeniorStudent.AcceptanceSteps*
  - *Install-Package NUnit -ProjectName SeniorStudent.AcceptanceSteps*
  - *Install-Package ShouldFluent -ProjectName SeniorStudent.AcceptanceSteps*
- Add the reference to your project following steps:
  - Ensure you are within the acceptance tests solution
  - Right click *References*
  - Click on *Add References*

  - In the left side column of the now visible Reference Manager, find and check the following item (make sure you actually click the check mark to the left)
    - *Solution->Projects->ProjectName*
    - Also, the reference to *nunit.framework* should have been added when the NUnit package was installed in the previous step, but if *nunit.framework* is not in the References list, be sure to add it. It can be found in the Reference Manager under *Assemblies->Extensions->nunit.framework*

- Finally, in an attempt to maintain good file structure, right click *SeniorStudent.AcceptanceSteps* then click on *Add* -> *New Folder* and name the new folder *StepDefinitions*. This will be where all Step Definition classes are held.

Also feel free to delete the auto-generated class *Class1.cs* as there is no need for it in a testing solution.

Feature Files

It is now time to add new Feature Files. In the *Solution Explorer* panel, right click then left click

SeniorStudent.AcceptanceSteps -> Add -> New item ->
Visual C# items (on left side) -> SpecFlow Feature File (5)

Name the Feature File *StudentHasAnExamSoon* then press *Add*. One will normally have many Feature Files, but in this simple example, there will only be one.

While in the *StudentHasAnExamSoon.feature* file, write the following code:

```
Feature: Student Has an Exam Soon
      In order to get through college
      As a student
      I must study and pass exams

Scenario: Student Has Exam Soon Without Studying
      Given I am a student
      And I have a textbook
      And I have not read the textbook
      When I study for the exam that is in five minutes
      Then I will die

Scenario: Student Takes Exam With Studying
      Given I am a student
      And I have a textbook
      And I have read the textbook
      When I study for the exam that is in five minutes
      Then I will be okay
```

To generate the step definitions (and realize how wonderful SpecFlow is), right click anywhere within the *StudentHasAnExamSoon.feature* feature file and click on

Generate Step Definitions -> Generate (6)

Save the file in the

SeniorStudent/SeniorStudent.AcceptanceSteps/StepDefinitions (7)

folder that was created earlier. *Note: in some cases when opening that file in Visual Studio, there will be an 'Inconsistent Line endings' error. Use the 'Windows (CR LF)' option and then press 'Yes'.*

It is now time for a trial run. The goal of this is see if *SeniorStudent.AcceptanceSteps* is able to talk to *SeniorStudent*. If all is correct so far, one will expect to see an error due to the quite obvious fact that the Step Definitions have not been filled in yet.

Build your project (shortcut is Ctrl+Shift+b). Go to the *Test Explorer* panel which can initially be opened with

TEST -> Windows -> Test Explorer (8)

if not already visible. In the Test Explorer panel, press *Run All*. If everything is correct so far, there will be results in *Not Run Tests* due to *One or more step definitions are not implemented yet.*

Fill in step definitions

This is the final step where one gets to write C# code.

Open the *StudentHasAnExamSoonStep.cs* class that was auto-generated in an earlier step. The task here is to replace the *ScenarioContext.Current.Pending();* calls with assertions and logical testing code. The syntax is rather simple once one sees a few examples.

First, be sure to add *using Should.Fluent;* to the top of the class for easier assertions.

Then fill in the Class to look like the following while paying attention to how SpecFlow uses *ScenarioContext.Current.Set()* and *ScenarioContext.Current.Get<>()* to pass objects through the steps outlines in the Feature File. Also note how the *Should.Fluent* package uses statements like *Student.IsAlive.Should().Be.False();* to do the assertions:

```
using System;
using TechTalk.SpecFlow;
using Should.Fluent;

namespace SeniorStudent.AcceptanceTests.StepDefinitions
{
    [Binding]
    public class StudentHasAnExamSoonSteps
    {
                [Given(@"I am a student")]
        public void GivenIAmAStudent()
        {
            var student = new Student();
            ScenarioContext.Current.Set(student);
        }

                [Given(@"I have a textbook")]
        public void GivenIHaveATextbook()
        {
            var textbook = new TextBook();
            ScenarioContext.Current.Set(textbook);
        }

                [Given(@"I have not read the textbook")]
        public void GivenIHaveNotReadTheTextbook()
        {
            var textbook =
                ScenarioContext.Current.Get<TextBook>();
            textbook.HasBeenRead = false;
            ScenarioContext.Current.Set(textbook);
        }

                [Given(@"I have read the textbook")]
        public void GivenIHaveReadTheTextbook()
        {
            var textbook =
                ScenarioContext.Current.Get<TextBook>();
            textbook.HasBeenRead = true;
            ScenarioContext.Current.Set(textbook);
        }

                [When(@"I study for the exam that is in five
                    minutes")]
        public void WhenIStudyForTheExamThatIsInFiveMinutes()
        {
            var student =
                ScenarioContext.Current.Get<Student>();
            var textbook =
                ScenarioContext.Current.Get<TextBook>();

            student.StudyForExam(textbook);
        }

                [Then(@"I will die")]
        public void ThenIWillDie()
        {
            var student =
                ScenarioContext.Current.Get<Student>();
            student.IsAlive.Should().Be.False();
        }

                [Then(@"I will be okay)")]
        public void ThenIWillBeOkay()
        {
            var student =
                ScenarioContext.Current.Get<Student>();
            student.IsAlive.Should().Be.True();
        }
    }
}
```

If everything has been done correctly, build your project, go to the Text Explorer panel, click *Run All*, and one should see a green check mark confirming all ran successfully.

If you want to explore further, feel free to create new Feature files, generate the step definitions, and fill in the C# Step Definition Class files.

References

[1] Wikipedia: Behavior-driven Development,
`http://en.wikipedia.org/wiki/Behavior-driven_development`

[2] Introducing BDD | Dan North and Associates,
`http://dannorth.net/introducing-bdd/`

[3] Gherkin Cucumber Wiki GitHub,
`https://github.com/cucumber/cucumber/wiki/Gherkin`

[4] BusinessReadableDSL,
`http://martinfowler.com/bliki/BusinessReadableDSL.html`