

## Introduction

- SpecFlow is ...
- This document describes the features and the usage of SpecFlow.
- To learn about the idea behind SpecFlow and about how to integrate it into the development process, see
  - BDD/ATDD: link
  - SpecFlow WorkFlow
  - Cucumber
  - Other practices & samples
- This document uses examples from the SpecFlow BookShop sample, that can be downloaded from here: <link>

## Installation

SpecFlow is distributed as a Windows Installer MSI file. In order to install it to your machine you have to execute this installer.

SpecFlow can integrate to Visual Studio 2008 and Visual Studio 2010. During the installation process you can decide whether to install the integration to the different Visual Studio versions.

The installer deploys the necessary files to the specified folder (default: *C:\Program Files\TechTalk\SpecFlow*) and registers the Visual Studio Integration.

## Execution of SpecFlow Tests

With SpecFlow you can define the acceptance criteria in .feature files that can be executed. These tests are usually placed in a separate project in the solution (e.g. "BookShop.AcceptanceTests" in the BookShop sample).

SpecFlow generated executable unit tests from the defined acceptance criteria (called scenarios). These generated unit tests are in the generated sub-items if the feature files (e.g. US01\_BookSearch.feature.cs).

The execution of the tests depends on the unit test provider used by SpecFlow (currently NUnit, MsTest and xUnit is supported). The unit test provider can be configured in the app.config file of the test project:

```
<specFlow>
  <unitTestProvider name="MsTest" />
</specFlow>
```

For example for MsTest unit test provider, the acceptance tests can be executed with the following steps:

1. Select the acceptance test project (e.g. "BookShop.AcceptanceTests") in solution explorer.
2. Select command from the main menu: Test / Run / Tests in Current Context (Ctrl R,T)

## Test Output and Result

When SpecFlow tests are executed the execution engine processes the test steps, executes the necessary test logic and either finish successfully or fails with some reason.

### Test Passes

While the tests are executed the engine also outputs useful information about the execution to the test output. Therefore in some cases it makes sense to investigate the test output even if the test was passing.

The test output shows by default the executed test steps, the invoked test logic methods (bindings) and the execution time of the longer operations. The information displayed in the test output can also be configured. See [<trace>](#) configuration element.

### Test Fails because of a Test/Business Logic Error

A test can fail because the test/business logic reports an error. This is reported as a test error, you can investigate the test output for the detailed information (e.g. stack trace) of the error.

### Missing, Pending or Improperly Configured Bindings

The test can also fail because some parts of the test logic (bindings) were not implemented yet (or configured improperly). This is reported by default with the “inconclusive” result. You can change how SpecFlow should behave in this case. See [<runtime>](#) configuration element.

Some unit test framework does not support inconclusive result. In this case the problem is reported as an error by default.

The test output can be very useful for missing bindings as it contain a step binding method skeleton that you can copy to your project and fill-in with the test logic.

### Ignored Tests

Just like with normal unit tests, you can also ignore a SpecFlow test. This can be done by marking the scenario with the `@ignore` [tag](#).

## Debugging Tests

SpecFlow Visual Studio integration also supports debugging the execution of the tests. Just like in the source code files of your project, you can also place breakpoints in the SpecFlow feature files.

Whenever you execute the generated tests in debug mode, the execution will stop at the specified breakpoints and you can execute the steps one-by-one using the standard “Step Over” (F10) command or you can go to the detailed execution of the bindings using the “Step Into” (F11) command.

Also when the execution of a SpecFlow test is stopped at a certain point of the binding (because of an exception for instance), you can navigate to the currently executed step of the feature file from the “Call Stack” tool window of Visual Studio.

## Technical Concept

- allows adding feature files to the projects (C#, VB.NET)

- the installed SpecFlow single-file generator generates a unit test when the feature file is saved
  - you can force generation from context menu: "Run Custom Tool"
- the generated unit test can be executed with the unit test execution tools
  - unit test provider has to be configured
  - the project type might depend on the selected unit test provider (e.g. Test Project for MsTest)
- the executed tests call out to the test logic ("bindings")
- the bindings can drive the application and implement the automation of the test steps
- SpecFlow: generator part and runtime part

## Setup SpecFlow Tests

Generally, just like for test-driven development, behavior-driven development works the best if it is integrated into the development process with a [test-first](#) approach and using an [outside-in](#) approach. Therefore it is important to setup the environment for the SpecFlow tests from the beginning of the application development.

Comment [GN1]: TODO: link

Comment [GN2]: TODO: link

The SpecFlow tests are usually placed into one or more separate project in the solution. Just like for unit tests, it makes sense to keep a consistent naming convention for these projects. The BookShop sample manages the acceptance criteria in a project called "BookShop.AcceptanceTests".

As mentioned in the [Technical Concept](#) section, SpecFlow uses a unit testing framework to execute the acceptance criteria defined in the feature files. Therefore setting up a project for the SpecFlow is very similar to set up a unit test project.

1. Create a new project that suits to the selected unit-testing framework (usually a *Class Library*, but for MsTest you need to create a *Test Project*).
2. Add a reference to the created project for the unit-test framework library (e.g. to `nunit.framework.dll` for NUnit).
3. Add a reference to the SpecFlow runtime assembly (`TechTalk.SpecFlow.dll`).
4. Add an "App.config" file to configure the unit test provider and other options of SpecFlow (see [Configuration](#)). This step is optional if you use NUnit.
5. Optional: Create a "StepDefinitons" folder for the step binding classes.
6. Optional: Create a "Support" folder for other infrastructural code (e.g. event bindings).

*Note: SpecFlow might provide later a Visual Studio project template to accomplish these steps.*

As the SpecFlow runtime assembly is not installed into the GAC, it is recommended to copy this assembly into the project source folder and store it in the source control together with your code. The other assemblies installed by SpecFlow are only required for the Visual Studio integration, reporting and other tools, so they are not necessarily required to keep together with your source.

## First Feature File

After configuring the project you can add the first feature files to your project (usually into the root folder of the project). This can be done using the Visual Studio item template (Add / New Item...), called "SpecFlow Feature".

This item template creates you a new feature file with a sample scenario. Change the file content according to the specification of your application and save the file. SpecFlow will generate the supporting unit test that you can execute to start the outside-in development process. See also:

[Missing, Pending or Improperly Configured Bindings](#)

You can read more about structuring the feature files and the bindings here: `<url>`

**Comment [GN3]:** TODO: url to cukes wiki

### Regenerate Unit Tests

Although normally working with the feature files in Visual Studio keeps the generated unit test code consistent with the feature file, in some cases you might want to re-run the generation (e.g. because you have modified the feature file outside of Visual Studio).

Re-generation can be forced by saving the files explicitly or doing the following steps:

1. Select the feature files in solution explorer (you can select multiple files within a project).
2. Invoke "Run Custom Tool" command from the context menu.

You can also configure the project to refresh the feature files (if necessary) before each compilation. Read more about this option in the [Update Generated Tests Automatically before Compilation using MsBuild](#) section. This might be useful if you regularly use another tool to edit the feature files.

## Upgrade Project to a Newer SpecFlow Version

TBD

- The cleanest way for upgrading is to install the new specflow installer (that replaces the visual studio integration so the generator part) and also replace the runtime part (TechTalk.SpecFlow.dll) in your project. After doing that re-generate the unit tests ([Regenerate Unit Tests](#)).
- The interface generator part and the runtime is usually compatible: the new runtime can run tests generated by the old generator and vica versa
- However, to take the advantage of the new generator features (or in case of a breaking change in the generator-runtime interface) you might need to regenerate all tests, see [Regenerate Unit Tests](#)

## Configuration

The behavior of SpecFlow can be extensively configured through .NET configuration files. SpecFlow processes the configuration file of the acceptance test projects (the projects that contain the feature files). The configuration has to be placed in a file called "App.Config" (the standard configuration file convention for .NET).

Unlike other runtime-only tools, SpecFlow processes the configuration file also while it generates the unit-tests from the feature files (this happens usually when you save the feature file). This means that after you have changed the configuration file, you might need to force re-generation of the unit test (if the configuration change affects the generated tests). See [Regenerate Unit Tests](#) for details about how this can be done.

## Default Configuration

In SpecFlow all configuration option has a default setting, so in an extreme case you don't need to specify any configuration file.

Commonly the most important thing to configure is the [unit test provider](#). Therefore simple SpecFlow projects configure only this aspect. The following example shows such a simple configuration.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="specFlow"
      type="TechTalk.SpecFlow.Configuration.ConfigurationSectionHandler,
        TechTalk.SpecFlow"/>
  </configSections>

  <specFlow>
    <unitTestProvider name="MsTest" />
  </specFlow>
</configuration>
```

The following example shows all possible configuration option with their default values.

```
<specFlow>
  <language feature="en-US" tool="" />

  <unitTestProvider name="NUnit" />

  <generator allowDebugGeneratedFiles="false" />

  <runtime detectAmbiguousMatches="true"
    stopAtFirstError="false"
    missingOrPendingStepsOutcome="Inconclusive" />

  <trace traceSuccessfulSteps="true"
    traceTimings="false"
    minTracedDuration="0:0:0.1"
    listener="TechTalk.SpecFlow.Tracing.DefaultListener,
      TechTalk.SpecFlow" />
</specFlow>
```

## Configuration Elements

### <language>

This section can be used to define the default language for the feature files and other language-related settings. Learn more about the language settings in the [Feature Language](#) section.

Attribute	Value	Description
<b>Feature</b>	culture name ("en-US")	The default language of the feature files added to the project. It is recommended to use specific culture names (e.g.: "en-US") and not generic (neutral) cultures (e.g.: "en"). Default: en-US

<b>Tool</b>	empty or culture name	<i>Specifies the language that SpecFlow uses for messages and tracing. Uses the default feature language if empty. (Currently only English is used for messages.)</i> Default: empty
-------------	-----------------------	---

#### <unitTestProvider>

This section can be used to specify the unit-test framework SpecFlow uses to execute the acceptance criteria. You can either use one of the built-in unit-test providers or you can specify the classes that implement the custom unit test providers.

Attribute	Value	Description
<b>name</b>	"NUnit", "MsTest" or "xUnit"	The name of the built-in unit test provider. If you specify this attribute, you don't have to specify the other two. Default: NUnit
<b>generatorProvider</b>	class name	An assembly qualified class name of a class that implements <i>TechTalk.SpecFlow.Generator.IUnitTestGeneratorProvider</i> interface.
<b>runtimeProvider</b>	class name	An assembly qualified class name of a class that implements <i>TechTalk.SpecFlow.IUnitTestRuntimeProvider</i> interface.

#### <generator>

This section can be used to specify various unit-test generation options.

Attribute	Value	Description
<b>allowDebugGeneratedFiles</b>	"true" or "false"	The debugger is by default configured to step through the generated code. This helps to debug from the feature files directly to the bindings (see <a href="#">Debugging Tests</a> ). This feature can be disabled by setting this attribute to "true". Default: false

#### <runtime>

This section can be used to specify various unit-test execution options.

Attribute	Value	Description
<b>detectAmbiguousMatches</b>	"true" or "false"	Specifies whether SpecFlow should report an error if there is an ambiguous match of step binding or just use the first one that matches. Default: true
<b>stopAtFirstError</b>	"true" or "false"	Specifies whether the execution should stop at the first error or should continue to try matching the subsequent steps (in order to detect missing steps). Default: false
<b>missingOrPendingStepsOutcome</b>	"Inconclusive", "Ignore" or "Error"	Specifies how SpecFlow should behave if a step binding is not implemented or pending. See <a href="#">Missing, Pending or Improperly Configured Bindings</a> . Default: Inconclusive

### <trace>

This section can be used to configure how and what should SpecFlow trace out to the unit test output.

Attribute	Value	Description
<b>traceSuccessfulSteps</b>	"true" or "false"	Specifies whether SpecFlow should trace successful step binding executions. Default: true
<b>traceTimings</b>	"true" or "false"	Specifies whether SpecFlow should trace execution time of the binding methods (only if the execution time is longer than the <i>minTracedDuration</i> value). Default: false
<b>minTracedDuration</b>	TimeSpan (0:0:0.1)	Specifies a threshold for tracing the binding execution times. Default: 0:0:0.1 (100 ms)
<b>Listener</b>	class name	An assembly qualified class name of a class that implements <i>TechTalk.SpecFlow.Tracing.ITraceListener</i> interface. SpecFlow provides <i>DefaultListener</i> and <i>NullListener</i> as default implementations. Default: <i>TechTalk.SpecFlow.Tracing.DefaultListener</i> , <i>TechTalk.SpecFlow</i>

## Gherkin Language Elements

The feature files that are used by SpecFlow to store the acceptance criteria of the features (use cases, user stories) of your application are described in a format that is called Gherkin. The Gherkin language defines the structure and a basic syntax for describing the tests. The Gherkin format is also used by other tools, like cucumber.

You can find a detailed description of the Gherkin language [here](#). In this document we only highlight elements in order to describe how SpecFlow handles them.

**Comment [GN4]:** Add link

### Features

The feature element provides the header or frame for the feature file. The feature has a title and a free-text high level description of the feature of your application that is detailed in the file.

SpecFlow generates a unit-test class for the feature element. The class name will be derived from the title of the feature.

### Scenarios

The feature file may contain multiple scenarios. The scenarios can be used to describe the acceptance tests of the feature. The scenario has a title and multiple scenario steps.

SpecFlow generates a unit test method for each scenario. The method name will be derived from the title of the scenario.

## Scenario Steps

The scenarios may contain multiple scenario steps. These steps describe the preconditions, the actions and the verification steps of the acceptance test (other terminologies are using arrange, act and assert for these parts). These steps are introduced with the *Given*, *When* and *Then* keywords (in English feature files), but subsequent steps of the same kind can be also specified with the *And* keyword. There may be other alternative keywords for specifying the steps, like the *But*.

The Gherkin syntax allows any combination or mixture of these three concepts, but usually the scenarios have a given, a when and a then *block* (set of steps).

The scenario steps are defined with a step text and can have additional table or multi-line text arguments.

SpecFlow generated a call inside the unit test method for each scenario step. The call is performed by the SpecFlow runtime that will execute the binding matching to the scenario step. The matching is done at runtime, so the generated tests can be compiled and executed even if the binding is not yet implemented. Read more about execution of test before the binding has been implemented in the section [Missing, Pending or Improperly Configured Bindings](#).

The scenario steps are primary way to execute any custom code to automate the application. You can read more about the different bindings in the [Bindings](#) section of this document.

## Table and multi-line text arguments

The scenario steps can have table or multi-line text arguments additionally to the step text (that can also contain arguments for the bindings, see [Step Bindings](#)). These are described in the subsequent lines of the scenario step and passed as additional *Table* and *string* arguments to the step bindings.

## Tags

Tags are markers that can be applied to features and scenarios. Applying a tag on a feature is equivalent to apply the tag to all scenarios in the feature file.

If the unit test framework supports it, SpecFlow generated categories from the tags. The generated category name is the tag name, without the leading @ sign. With the generated unit test categories you can filter or group the tests being executed. For example by marking crucial test with the *@important* tag, you can execute these tests more frequently.

Even if the unit test framework does not support categories, you can use the tags to execute special logic for the tagged scenarios in event bindings ([Event Bindings](#)) or in the step bindings by investigating the *ScenarioContext.Current.ScenarioInfo.Tags* property.

The *@ignore* tag is handled specially by SpecFlow. From the scenarios marked with this tag SpecFlow generates an ignored unit test method. See [Ignored Tests](#).

## Background

The background language element allows specifying a common precondition for all scenarios in a feature file. The background part of the file can contain one or more scenario steps that are executed before any other steps of the scenarios.



SpecFlow generates a method from the background elements that is invoked from all unit tests generated for the scenarios.

## Scenario Outlines

Scenario outlines can be used to define data-driven acceptance tests. They can be also seen as scenario templates. The scenario outline is always consisting of a scenario template specification (a scenario with data placeholders using the `<placeholder>` syntax) and a set of examples that provide values for the placeholders.

SpecFlow generates a parametrized unit test logic method for a scenario outline and an individual unit test method for each example set.

For better traceability, the generated unit test method names are derived from the scenario outline title and the first value of the examples (first column of the examples table). Therefore it is a good practice to choose a unique and descriptive parameter as the first column of the example set. As the Gherkin syntax does not enforce that all example columns have the matching placeholder in the scenario outline, you can even introduce an arbitrary column in the example sets for better test method name readability.

SpecFlow performs the placeholder substitution as a separate phase before the step binding match would be applied. Therefore the implementation and the parameters of the step bindings are independent of whether they are executed through a direct scenario or a scenario outline. This leaves the option to specify further examples to the acceptance tests later without changing the step bindings.

## Feature Language

To avoid communication errors introduced by translations, it is recommended to keep the specification and the acceptance test descriptions in the language of the business. The Gherkin format supports many natural languages besides English, like German, Spanish or French. You can find the list of all supported languages [here](#).

Comment [GN5]: Add link

The language of the feature files can be either specified globally in the configuration (see [<language>](#) element) or in the header of the feature file with the `#language` syntax. The language has to be specified using the ISO language names used by the `CultureInfo` class of the .NET Framework (like `en-US`).

```
#language: de-DE
Funktionalität: Addition
...
```

SpecFlow uses the feature file language to use the right set of keywords when parsing the file, but the language setting is also used when any parameter conversion has to be done by the SpecFlow runtime. As data conversion can only be done using a [specific culture](#) in the .NET Framework it is recommended to use the specific culture name (`en-US`) instead of the [neutral culture](#) names (`en`). If a neutral culture is used, SpecFlow uses a default specific culture for data conversions (e.g. uses the `en-US` for conversion if the `en` language was used).

Comment [GN6]: Add link to msdn

Comment [GN7]: Add link to msdn

## Comments

You can add comment lines to the feature files at any place starting the line with the # sign. Be careful however, as comments in the specification are often signs of wrongly specified acceptance criteria.

The comment lines are ignored by SpecFlow.

## Bindings

### Step Bindings

### Event Bindings

### Argument Conversions

#### Simple Conversions

#### Step Argument Transformations

## Communication between Bindings

### Instance Fields

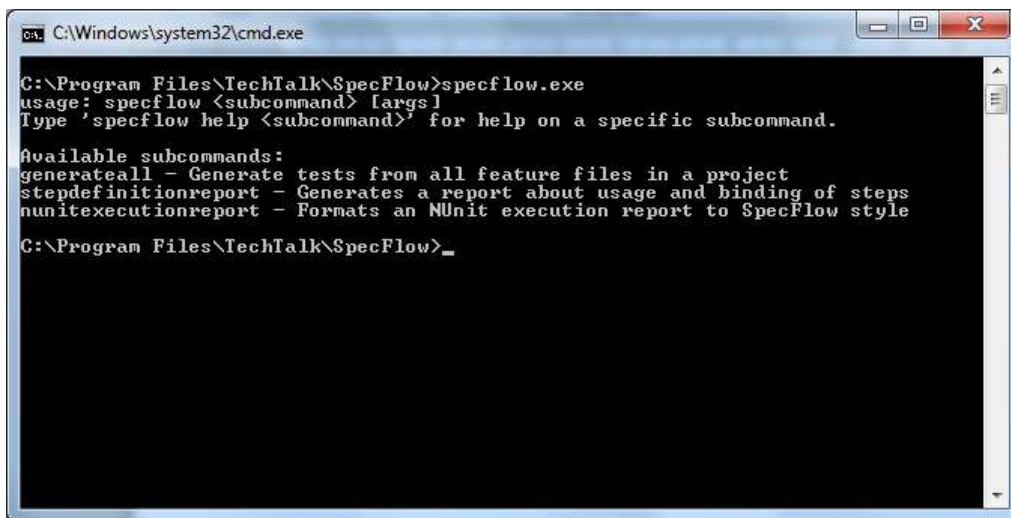
### Static Fields

### ScenarioContext and FeatureContext

### Context Injection

## Tools

Besides the execution of the acceptance criteria of the application, SpecFlow supports the development process with further useful tools. This additional tools can be invoked through the specflow.exe command-line tool. The tool has to be parametrized from command line arguments. Executing the tool without any argument displays the possible commands. The “help” command can be used to display the detailed usage of the specific commands.



```
C:\Windows\system32\cmd.exe

C:\Program Files\TechTalk\SpecFlow>specflow.exe
usage: specflow <subcommand> [args]
Type 'specflow help <subcommand>' for help on a specific subcommand.

Available subcommands:
generateall - Generate tests from all feature files in a project
stepdefinitionreport - Generates a report about usage and binding of steps
nunitexecutionreport - Formats an NUnit execution report to SpecFlow style

C:\Program Files\TechTalk\SpecFlow>_
```

One part of the available tools are to provide different reports. The section [Reporting](#) discusses these options.

### Generate All

The “generateall” command can be used to re-generate all outdated unit test classes based on the feature file. This tool can be useful when upgrading to a newer SpecFlow version or when the feature files are modified outside of Visual Studio.

The following table contains the possible arguments for this command.

Argument	Value	Description
<b>projectFile</b>	File path	A path of the project file containing the feature files. This is a mandatory argument.
<b>/force</b>	-	Forces re-generation even if the generated class is up-to-date based on the file modification time. Default: disabled (only outdated files are generated)
<b>/verbose</b>	-	Displays detailed information about the generation process. Default: disabled

The following example shows how to regenerate the unit tests for the BookShop sample application.

```
specflow.exe generateall BookShop.AcceptanceTests.csproj
```

### Update Generated Tests Automatically before Compilation using MsBuild

The “generate all” command can be also invoked from MsBuild. This way the unit test files can be updated before compiling the solution. This can be useful if the feature files are regularly modified outside of Visual Studio.

In order to enable this in your project, you have to modify the project file containing the feature files (e.g. with notepad). You have to add only one line to the end of project file as the following example shows.

```
...
</ItemGroup>
<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
<Import
  Project="$(ProgramFiles)\TechTalk\SpecFlow\TechTalk.SpecFlow.targets"/>
</Project>
```

In order to be able to build your application in any environment independent of the SpecFlow installation it is recommended to store the SpecFlow tools together with your sources and use a relative path for the import.

```
<Import Project="..\lib\SpecFlow\TechTalk.SpecFlow.targets"/>
```

The TechTalk.SpecFlow.targets file can be investigated for further possibilities of calling this command from MsBuild.

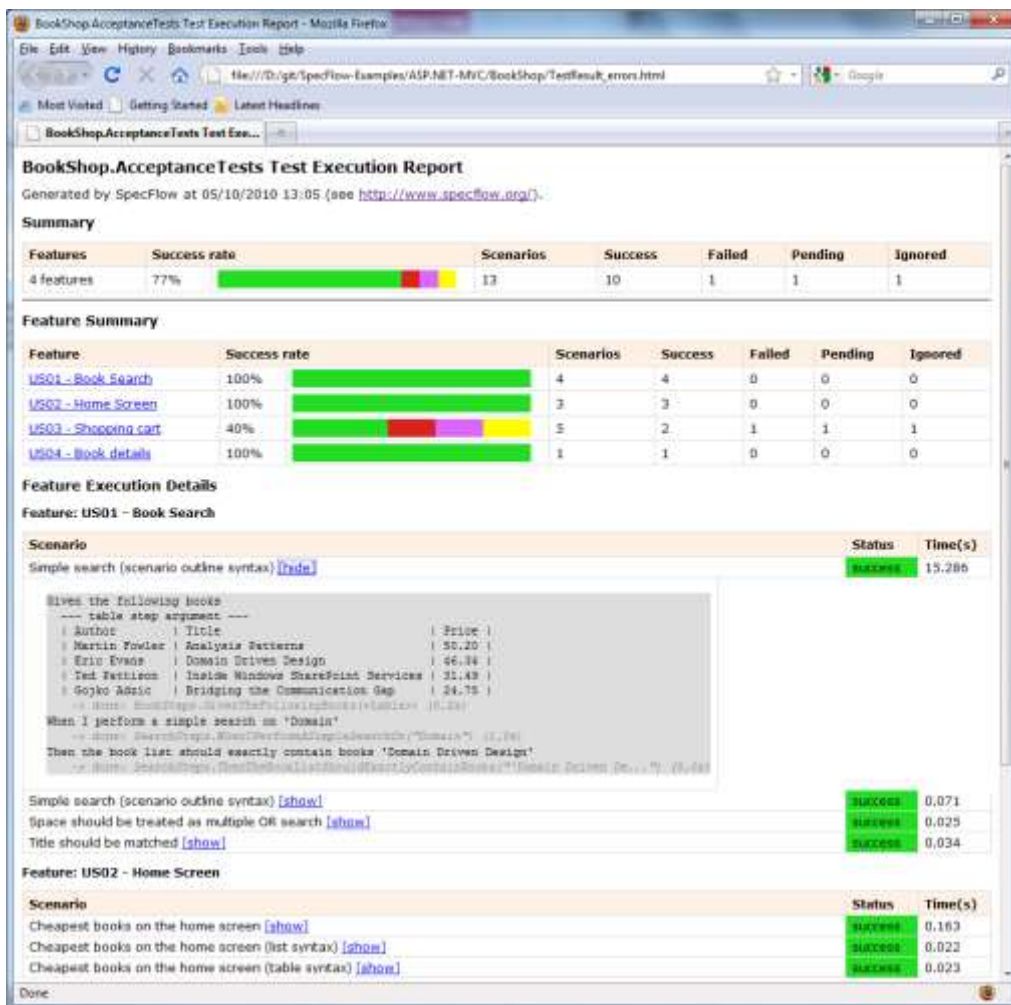
## Reporting

SpecFlow provides various options to generate reports related to the acceptance tests.

### NUnit Test Execution Report

This report provides a formatted HTML report of a test execution if the NUnit test provider was used. Similar execution reports will be available for the other unit-test providers soon.

The report contains a summary about the executed tests and the result and also a detailed report for the individual scenario executions.



In order to generate this report you have to execute the acceptance tests with the *nunit-console* runner. This tool generates an XML summary about the test executions. To have the detailed scenario execution traces visible, you also need to capture the test output using the */out* and the */labels* options as it can be seen in the following example.

```
nunit-console.exe /labels /out=TestResult.txt /xml=TestResult.xml
bin\Debug\BookShop.AcceptanceTests.dll
```

The two generated file can be used to invoke the SpecFlow report generation. If you use the output file names shown above it is enough to specify the project file path containing the feature files.

```
specflow.exe nunitexecutionreport BookShop.AcceptanceTests.csproj
/out:MyResult.html
```

**TODO: arguments table**

## Step Usage Report

specflow.exe stepdefinitionreport BookShop.AcceptanceTests.csproj
---