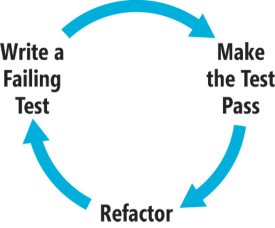
SpecFlow

1. **A Brief History of Automated Testing**

One of the most valuable practices to emerge from the Agile Software movement is an automated, test-first development style, often referred to as Test-Driven Development, or TDD. A key tenet of TDD is that test creation is as much about design and development guidance as it is about verification and regression. It’s also about using the test to specify a unit of required functionality, and using that test to then write only the code needed to deliver that functionality. Therefore, the first step in implementing any new functionality is to describe your expectations with a failing test (see Figure 1).



*Figure 1 The Test-Driven Development Cycle*

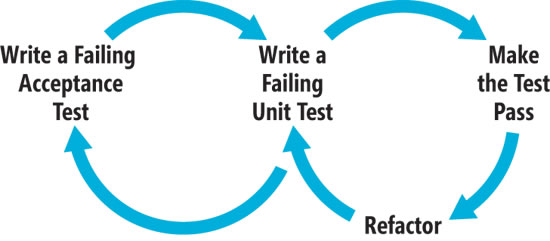
Many developers and teams have had great success with TDD. Others have not, and find that they struggle with managing the process over time, especially as the volume of tests begins to grow and the flexibility of those tests begins to degrade. Some aren’t sure how to start with TDD, while others find TDD easy to initiate, only to watch it abandoned as deadlines near and large backlogs loom. Finally, many interested developers meet resistance to the practice within their organizations, either because the word “test” implies a function that belongs on another team or because of the false perception that TDD results in too much extra code and slows down projects.

Steve Freeman and Nat Pryce, in their book, “Growing Object-Oriented Software, Guided by Tests” (Addison-Wesley Professional, 2009), note that “traditional” TDD misses some of the benefits of true test-first development:

“It is tempting to start the TDD process by writing unit tests for classes in the application. This is better than having no tests at all and can catch those basic programming errors that we all know but find so hard to avoid … But a project with only unit tests is missing out on critical benefits of the TDD process. We’ve seen projects with high-quality, well unit-tested code that turned out not to be called from anywhere, or that could not be integrated with the rest of the system and had to be rewritten.”

In 2006, Dan North documented many of these challenges in an article in Better Software magazine (dannorth.net/introducing-bdd). In his article, North described a series of practices that he had adopted over the prior three years while in the trenches with testing. While still TDD by definition, these practices led North to adopt a more analysis-centric view of testing and to coin the term Behavior-Driven Development to encapsulate this shift.

One popular application of BDD attempts to extend TDD by tightening the focus and process of creating tests through Acceptance Tests, or executable specifications. Each specification serves as an entry point into the development cycle and describes, from the user’s point of view and in a step-by-step form, how the system should behave. Once written, the developer uses the specification and their existing TDD process to implement just enough production code to yield a passing scenario (see Figure 2).



*Figure 2 The Behavior-Driven Development Cycle*

**Where Design Begins**

BDD is considered by many a superset of TDD, not a replacement for it. The key difference is the focus on initial design and test creation. Rather than focusing on tests against units or objects, as with TDD, I focus on the goals of my users and the steps they take to achieve those goals. Because I’m no longer starting with tests of small units, I’m less inclined to speculate on fine-grained usage or design details. Rather, I’m documenting executable specifications that prove out my system. I still write unit tests, but BDD encourages an outside-in approach that starts with a full description of the feature to be implemented.

Let’s look at an example of the difference. In a traditional TDD practice, you could write the test in Figure 3 to exercise the Create method of a CustomersController.

[Test]

public void PostCreateShouldSaveCustomerAndReturnDetailsView()

{

    var customersController = new CustomersController();

    var customer = new Customer

    {

        Name = "Hugo Reyes",

        Email = "hreyes@dharmainitiative.com",

        Phone = "720-123-5477"

    };

    var result = customersController.Create(customer) as ViewResult;

    Assert.IsNotNull(result);

    Assert.AreEqual("Details", result.ViewName);

    Assert.IsInstanceOfType(result.ViewData.Model, typeof(Customer));

    customer = result.ViewData.Model as Customer;

    Assert.IsNotNull(customer);

    Assert.IsTrue(customer.Id > 0);

}

*Figure 3 Unit Test for Creating a Customer*

With TDD, this tends to be one of the first tests I write. I design a public API to my CustomersController object by setting expectations of how it will behave. With BDD I still create that test, but not at first. Instead, I elevate the focus to feature-level functionality by writing something more like Figure 4. I then use that scenario as guidance toward implementing each unit of code needed to make this scenario pass.

Feature: Create a new customer

*In order to improve customer service and visibility*

*As a site administrator*

*I want to be able to create, view and manage customer records*

Scenario: Create a basic customer record

  Given I am logged into the site as an administrator

  When I click the "Create New Customer" link

  And I enter the following information

    |*Field*|*Value*|

    | Name  | Hugo Reyes                  |

    | Email | hreyes@dharmainitiative.com |

    | Phone | 720-123-5477                |

  And I click the "Create" button

  Then I should see the following details on the screen:

    |*Value*|

    | Hugo Reyes                  |

    | hreyes@dharmainitiative.com |

    | 720-123-5477                |

*Figure 4 Feature-Level Specification*

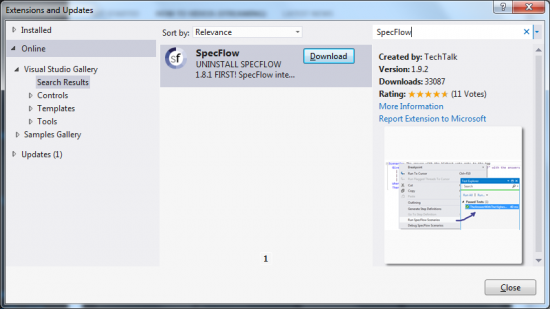
This is the outer loop in Figure 2, the failing Acceptance Test. Once this test has been created and fails, I implement each step of each scenario in my feature by following the inner TDD loop depicted in Figure 2. In the case of the CustomersController in Figure 3, I’ll write this test once I reach the proper step in my feature, but before I implement the controller logic needed to make that step pass.

1. **Quick Start**

SpecFlow aims at bridging the communication gap between domain experts and developers by binding business readable behavior specifications and examples to the underlying implementation. Its mission is to provide a pragmatic and frictionless approach to Specification-By-Example for .NET projects. SpecFlow also supports the concepts of Acceptance Test Driven Development (ATDD) and Behavior Driven Development (BDD), which are often used synonymously with Specification-By-Example.

* 1. **INSTALLATION**
     1. **Install IDE** **Integration**

For Visual Studio 2013, 2012 and 2010, the easiest way is to search for “SpecFlow” in the extension manager (Extensions and Updates in VS2013/VS2012) online search.



* + 1. **Setup SpecFlow Project**

The SpecFlow tests are usually placed into one or more separate project in the solution: the “SpecFlow Projects”. The easiest and most convenient way to setup these projects is to use our NuGet package: [SpecFlow](http://www.nuget.org/packages/SpecFlow) or one of the specific [helper packages](http://newweb.specflow.org/documentation/NuGet-Integration/), like [SpecFlow.NUnit](http://www.nuget.org/packages/SpecFlow.NUnit) or [SpecRun.SpecFlow](http://www.nuget.org/packages/SpecRun.SpecFlow). For a detailed project setup guide, check the [Setup SpecFlow Projects](http://www.specflow.org/documentation/Setup-SpecFlow-Projects/) page. If your application does not use NuGet (such as myTE), please add reference of SpecFlow dll file for your testing project (SpecFlow.1.6.1\lib\net35\TechTalk.SpecFlow.dll is used in myTE).

* 1. **FIRST STEPS**

This section guides you through the first steps of setting up a SpecFlow project and creating the first scenario. In the example, we use SpecFlow with NUnit, but you can choose from many other test execution frameworks, including [SpecFlow+Runner](http://www.specflow.org/plus/) that provides specialized execution features, Visual Studio Test Window and Team Foundation Server (TFS) Build integration for SpecFlow.

Have your first SpecFlow test running in just a few steps:

### Create a class library project: MyProject.Specs

If you want to use SpecFlow with MsTest in VS2010, you should start with a “Unit Test Project”

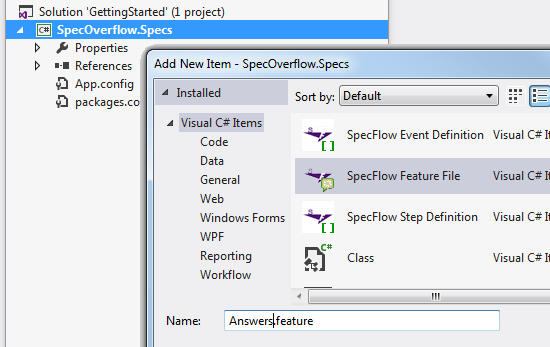
### Add SpecFlow for NUnit through NuGet

instead of NUnit you can also use [other test engines](http://go.specflow.org/doc-unittestproviders), like MsTest, xUnit, MbUnit or [SpecFlow+Runner](http://www.specflow.org/plus/)

PM> Install-Package SpecFlow.NUnit

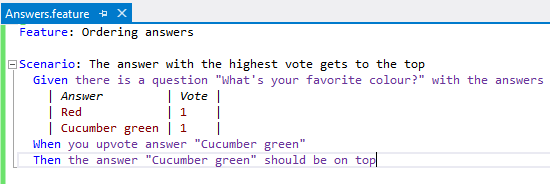
### Add your first feature file

we write yet-another StackOverflow clone now: [SpecOverflow](http://www.specflow.org/media/GettingStarted-SpecOverflow.jpg)



Add new feature file to the project

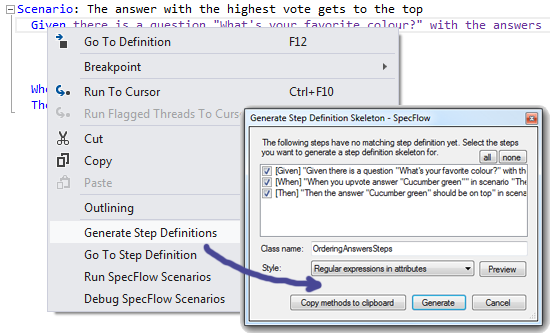
### Describe the first scenario of your application



The first scenario

### Generate step definition skeletons

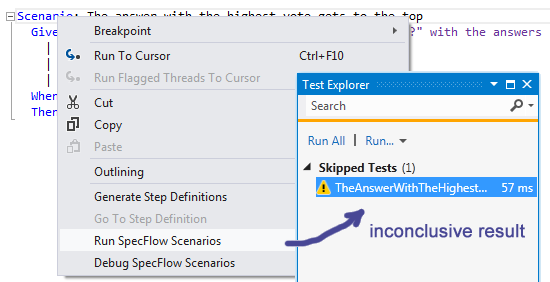
Cool new feature in v1.9!



Invoke step definition skeleton generation from the context menu

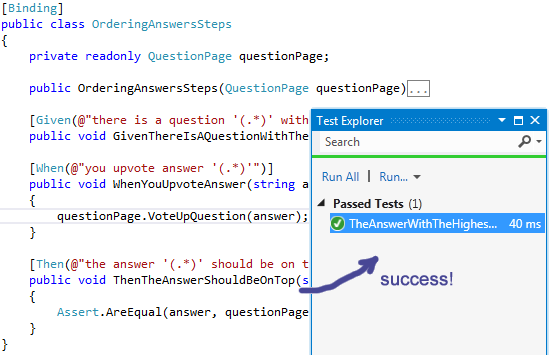
### Run your test – it is not green yet

Use your favorite test runner, like the VS2012 test runner below; but it works well with ReSharper too…



Running the test produces inconclusive result

### Implement automation & application logic until the test is green



The test finally passes

### Celebrate the good work and continue with the next scenario!

1. **SpecFlow in myTE**

MyTE is the first project using SpecFlow to implement BDD methodology in Accenture CIO. It reduce the gap of understanding in different ways between function team and development team. So that development team can stand on the same page at the very beginning of one feature or one user story before developing. And it indeed prevents unproductive effort.

There are almost 30 folders in which contains Scenarios to cover most of myTE business. The steps used by those Scenarios are defined in Steps folder.

* 1. **Steps**

1. **CommonStepDefinition**

It is used to define some common steps such as:

Initialization: Given (@"a new scenario")

Login: common login, represent, etc.

Set user preference: set reviewer, approver, home location, etc.

Common TR operation: save, submit, perform PPA, send for approval, request update, process TR, etc.

Common result or error: have or have no validation error, etc.

1. **StepDefinitionBase**

There is a singleton instance of StepDefinitionBase in class CommonStepDefinition. 2 builder class instance could be found in it. They are TimeReportBuilder and ExpenseBuilder which are used for basic TR and Expense setup.

1. **Other steps**: TimeEntry, Expense, Assignment, Audit, Review, Authorization, CarInformation, PunchClock, etc.
   1. **Builders**
2. **TimeReportBuilder**

Used for TR creation, saving, querying

All services and controller used in Builder are retrieved by name using GetObjectFromSpringContext method of ContextProvider. DB will be cleared (data will be deleted) if having step [Given(@"I have (.\*) with enterpriseid (.\*) into period (.\*)")]

1. **ExpenseBuilder**

Used for expense creation, saving, querying

* 1. **Example:**

Scenario: Going to expenses tab for reviewee through Reviewer functionality

Given a new scenario

And I have *login* with enterpriseid *user.ar.fulltime* into period *01/15/2014*

And I have created an expense *EX07* for country *AR* to regular project *SPECIAL*

And I have set the following expense values

|*ElementName*|*Value*|

| T2\_CountryOfExpense | AR               |

| T2\_Type             | healthInsurance  |

| T2\_Descriptions     | Description text |

| Amount              | 50               |

When I press save expense into the database

And I press submit timereport

And timereport is saved into database

Given I changed to login with enterpriseid *myte0211* into period *01/15/2014*

When I select *user.ar.fulltime* from my *reviewees* list

And I go to expenses tab

Then I see the following expense list

|*SequenceNumber*|*Description*|*Amount*|

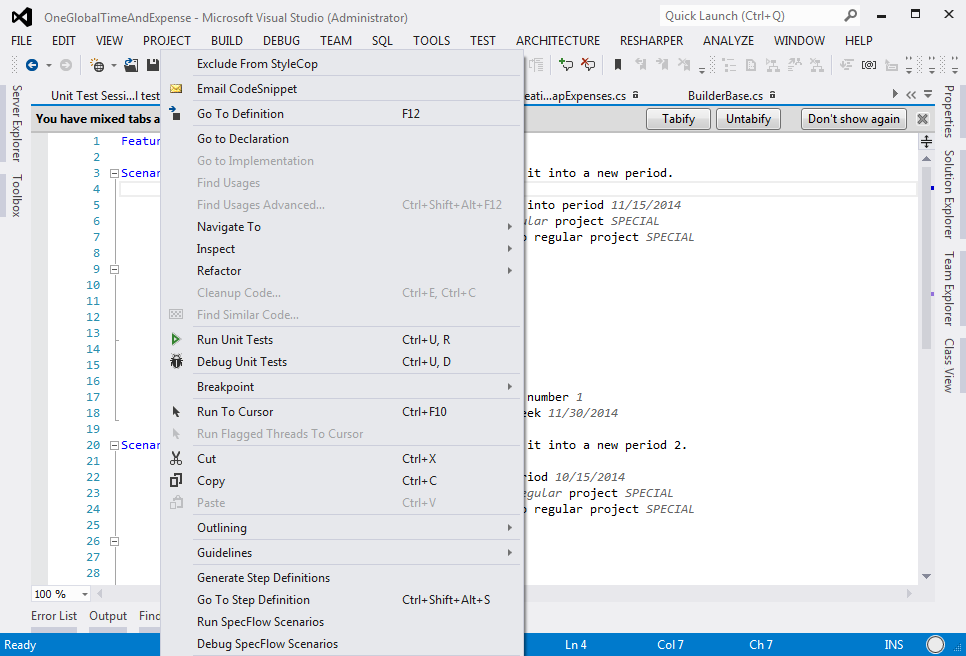
| 1              | Other Expense | 50     |

The example above is a simple test case for reviewing feature. At first, simulate login using user.ar.fulltime for period 01/15/2014, then create a expense charge with some values. And then save and submit this TR. Change login using another id, and review the TR created previously.

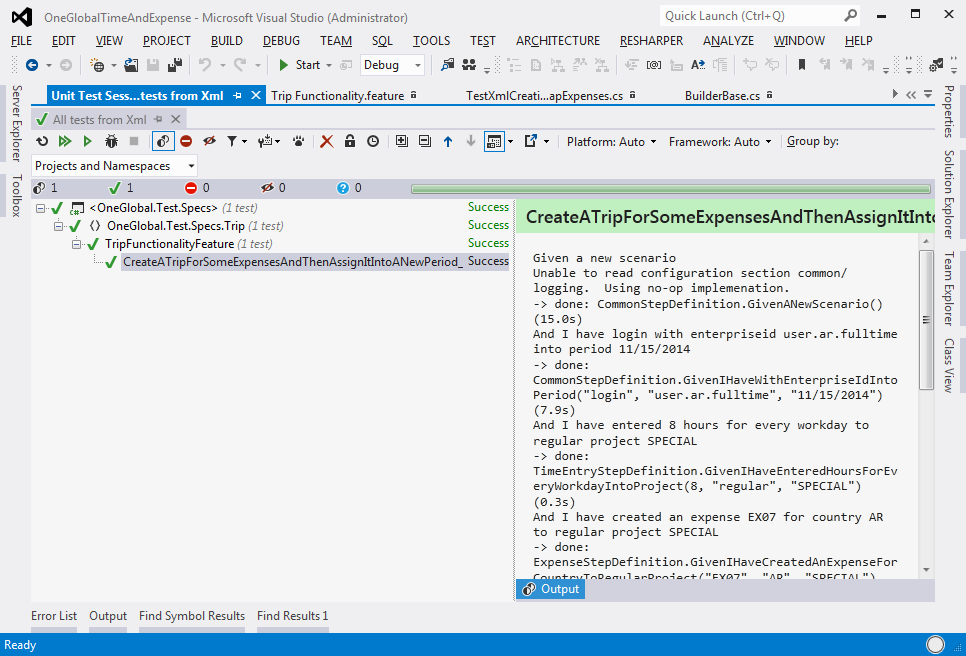
* 1. **How to test it?**

Because myTE uses NUnit as testing framework, you can get find testing result in NUnit test window. If you already installed ReSharper, it would be easier to you to manager testing in Unit Test Sessions window. There are 2 ways to run/debug SpecFlow.

1. Run/Debug SpecFlow Scenarios in feature file
2. Run/Debug SpecFlow Scenarios in solution explorer



You can get output from SpecFlow in order to diagnose your test if any issue.



* 1. **Some tips**

1. Please use TransactionTemplate if you need to get AccentureUser from service. I can avoid lazy load issue.
2. If new testing data is needed in DB, please modify excel file TestData.xls. And then run DataLoader to import data in to DB.
3. **References**

<http://en.wikipedia.org/wiki/Behavior-driven_development>

<http://dannorth.net/introducing-bdd/>

<http://www.specflow.org/documentation/>

<http://msdn.microsoft.com/zh-tw/library/dn743856.aspx> Step by step guide of TDD & BDD - Chinese version only