Behavior Driven Development (BDD), A way to document and test your features in a human-readable language

Ibrahim Abdelkareem (iaiab@kth.se)
2023-04-10

Software development is often deluged by overwork, which translates into wasted time and resource for both engineers and business teams. Communication between both parties is often the bottleneck to project progress when engineers often misunderstand what the business truly needs from its software, and business professionals misunderstand the capabilities of their technical team.

Shrouded in miscommunication and complexity, the end product is often technically functional, but fails to meet the exact requirements of the business. **Behaviour-driven development** aims to change this.

This essay is going to help you understand **BDD** and how it can best serve you. Before we jump into that, I'll briefly demonstrate some prerequisite knowledge and terminology necessary to understand BDD.

Prerequisite Knowledge

Acceptance Testing

A method of software testing where a system is tested for acceptability. The major aim of this test is to evaluate the compliance of the system with the business requirements and assess whether it is acceptable for delivery or not.

Acceptance Testing is the last phase of software testing performed after System Testing and before making the system available for actual use [4].

Test-Driven Development (TDD)

A process in which test cases are written before the code that validates those cases. It depends on repetition of a very short development cycle. Test driven Development is a technique in which automated Unit test are used to drive the design and free decoupling of dependencies. The following sequence of steps is generally followed when doing TDD:

1. Understand the features and requirements using user stories and use cases.
2. Write a test case that describe the function completely.

3. Run all the test cases and make sure that the new test case fails.
4. Write the code that passes the test case
5. Run the test cases
6. Refactor code – This is done to remove duplication of code.
7. Repeat the above-mentioned steps.

User Stories

User Story refers to a short, informal, and simple description of software features that are required by the end-users in the software system. Its main purpose is to provide software features that will add value to the customer requirements. User stories are considered an important tool in Incremental software development. Mainly a user story defines the type of user, their need, and why they need that.

User stories are completely from the end-user perspective which is defined based on the Role-Feature-Benefit pattern.

```
As an <actor>
I want a <feature>
So that <benefit>
```

E.g.,

```
As an mobile bank customer
I want to see balance on my accounts
So that I can make better informed decisions about my spending
```

Behavior-driven development (BDD)

Behavior-Driven Development (BDD) refers to an Agile software development process that is derived from the TDD methodology. BDD is considered as a test to illustrate the behavior of the system. It encourages the use of conversation and concrete examples in simple language for everyone involved in the development to bring better clarity to the behavior of the system. In this development, techniques define various ways to develop a feature based on the system's behavior, and the techniques are combined from Test Driven Development (TDD) and Domain Driven Development (DDD).

BDD is a good approach in Automated Testing as it is more focused on the behavior of the system rather than the implementation of the code. BDD is facilitated through natural language to express the behavior of the system and the expected outcomes from the system. In BDD all parties are involved (e.g., customer, developer, tester, and stakeholder) for a collaborated conversation and illustration of the system's behavior.

BDD Life-Cycle

The BDD lifecycle can be described in the following sequence:

1. Describe Behavior
2. Define Requirements
3. Run & Fail the tests
4. Apply code updates
5. Run & Pass the tests

You can see the steps to doing BDD are almost the same as for TDD. However, it’s fundamentally different as BDD isn’t about testing like TDD. It’s about describing behavior and achieving business goals and requirements.

At the beginning, the teams define the desired behavior of the system. This includes the flow and features of the product as user stories.

Then, the teams define more granular requirements for each user story to provide a shared understanding of the features. In BDD, this is done by defining requirements for each feature as a series of acceptance tests in a common format known as Given-When-Then. This format is also known as the Gherkin syntax among technical people.

```
Given some initial context (the givens),
When an event occurs,
Then ensure some outcomes.
```

E.g.,

```
Scenario: Show balance on the accounts page after logging in

  Given I have just logged on to the mobile banking app
  When I load the accounts page
  Then I can see account balance for each of my accounts


Scenario: Do not show balance if not logged in

  Given I am not logged on to the mobile banking app
  When I open the mobile banking app
  Then I can see a login page
  And I do not see account balance
```

Using gherkin serves multiple purposes:

- Unambiguous executable specification
- Document how the system actually behaves
- Automated testing using BDD testing frameworks (e.g., Cucumber)

Advantages of BDD

Using BDD to build software comes with many advantages such as:

- Strong Collaboration: Thanks to the shared language, product owners, developers, and testers all have in-depth visibility of the project's progression.
- Shorter Learning Curve: Since BDD is explained using very simple language, there's a much shorter learning curve for everyone involved.
- High Visibility: Being a nontechnical process by nature, behavior-driven development can reach a much wider audience.
- Rapid Iterations: BDD allows you to respond quickly to any and all feedback from your users so you can make improvements to meet their needs.
- BDD Test Suite: Similar to adopting TDD, adopting BDD gives your team confidence in the form of a test suite.
- Eliminate Waste: BDD allows you to clearly communicate requirements so there's less rework due to misinterpreted requirements and acceptance criteria.
- Focus on User Needs: Satisfied users are good for business and the BDD methodology allows user needs to be met through software development
- Meet Business Objectives: With BDD, each development can be traced back to the actual business objectives.

Now that we know how to describe system behavior as a series of acceptance tests written in a human-readable way that doesn't require technical knowledge in order to understand. The next step is to make these specifications executable.

There are many tools to bind the gherkin syntax to executable automated tests. The most popular tool is cucumber.

Introducing Cucumber

Cucumber reads executable specifications written in plain text and validates that the software does what those specifications say. The specifications consists of multiple examples, or scenarios.

It supports most (If not all) programming languages:

- Cucumber.js: JavaScript / TypeScript
- SpecFlow: .NET (C#, F#, VB.NET)
- Cucumber-JVM: Java

Using cucumber is easy:

1. First, Developers write the gherkin documents in **.feature** text files, that're typically versioned in source-control (e.g., git) alongside the code.
2. Then, Developers define **Step Definition** for each step (e.g., given, when, or then) in code.

Step Definition

Step definitions connect Gherkin steps to programming code. A step definition carries out the action that should be performed by the step. So step definitions hard-wire the specification to the implementation.

Example

Let’s assume we’re building a calculator, and the business wants to add the division feature to the calculator. The teams will start first by defining the user story.

```
As a calculator user
I want a division feature
So that I can divide two numbers
```

Then the teams will refine the user story and define more granular requirements.

```
Scenario: Simple Division
  Given the first number is 6
  And the second number is 2
  When the two numbers are divided
  Then the result should be 3
```

But that’s not all! There are also some edge-cases that the teams are going to discuss and define.

```
Scenario: Division by 0
  Given the first number is 6
  And the second number is 0
  When the two numbers are divided
  Then an error message show to the user with "Can't divide by 0"
```

Of course, this is a simple feature. A more real-life example feature will have much more different scenarios. Sticking to our simple example, the result gherkin file will look like that.

```
Title: Division

As a calculator user
I want a division feature
So that I can divide two numbers

Scenario: Simple Division
  Given the first number is 6
  And the second number is 2
  When the two numbers are divided

Scenario: Division by 0
  Given the first number is 6
  And the second number is 0
  When the two numbers are divided
  Then an error message show to the user with "Can't divide by 0" Then the result sh
```

Then, developers will define step definitions like the following example (This example is in C# using SpecFlow):

```
using FluentAssertions;
using TechTalk.SpecFlow;

namespace SpecFlowCalculator.Specs.Steps
{
    [Binding]
    public sealed class CalculatorStepDefinitions
    {
        // For additional details on SpecFlow step definitions see https://go.specfl

        private readonly ScenarioContext _scenarioContext;

        private readonly Calculator _calculator = new Calculator();
        private int _result;

        public CalculatorStepDefinitions(ScenarioContext scenarioContext)
        {
            _scenarioContext = scenarioContext;
        }

        [Given("the first number is (.*)")]
        public void GivenTheFirstNumberIs(int number)
        {
            _calculator.FirstNumber = number;
        }

        [Given("the second number is (.*)")]
        public void GivenTheSecondNumberIs(int number)
        {
            _calculator.SecondNumber = number;
        }

        [When("the two numbers are divided")]
        public void WhenTheTwoNumbersAreDivided()
        {
            _result = _calculator.Divide();
        }

        [Then("the result should be (.*)")]
        public void ThenTheResultShouldBe(int result)
        {
            _result.Result.Should().Be(result);
        }

        [Then("an error message show to user with (.*)")]
        public void ThenAnErrorMessageShowToTheUser(string errorMessage)
        {
            _result.ErrorMessage.Should().Be(errorMessage);
        }
    }
}
```

As you see from the example above, The attributes denoted by `[]` are used to bind gherkin steps to methods in code (Step definitions). SpecFlow (a Cucumber tool) is used to read the code and gherkin files and match step definitions with steps.

In the code-behind you can use Regex also to define parameters. Imagine now I would like to add another scenario with a step that says `The result should be 55` , We can re-use the same step defined for the aforementioned scenario if the number is a parameter.

Running BDD Tests

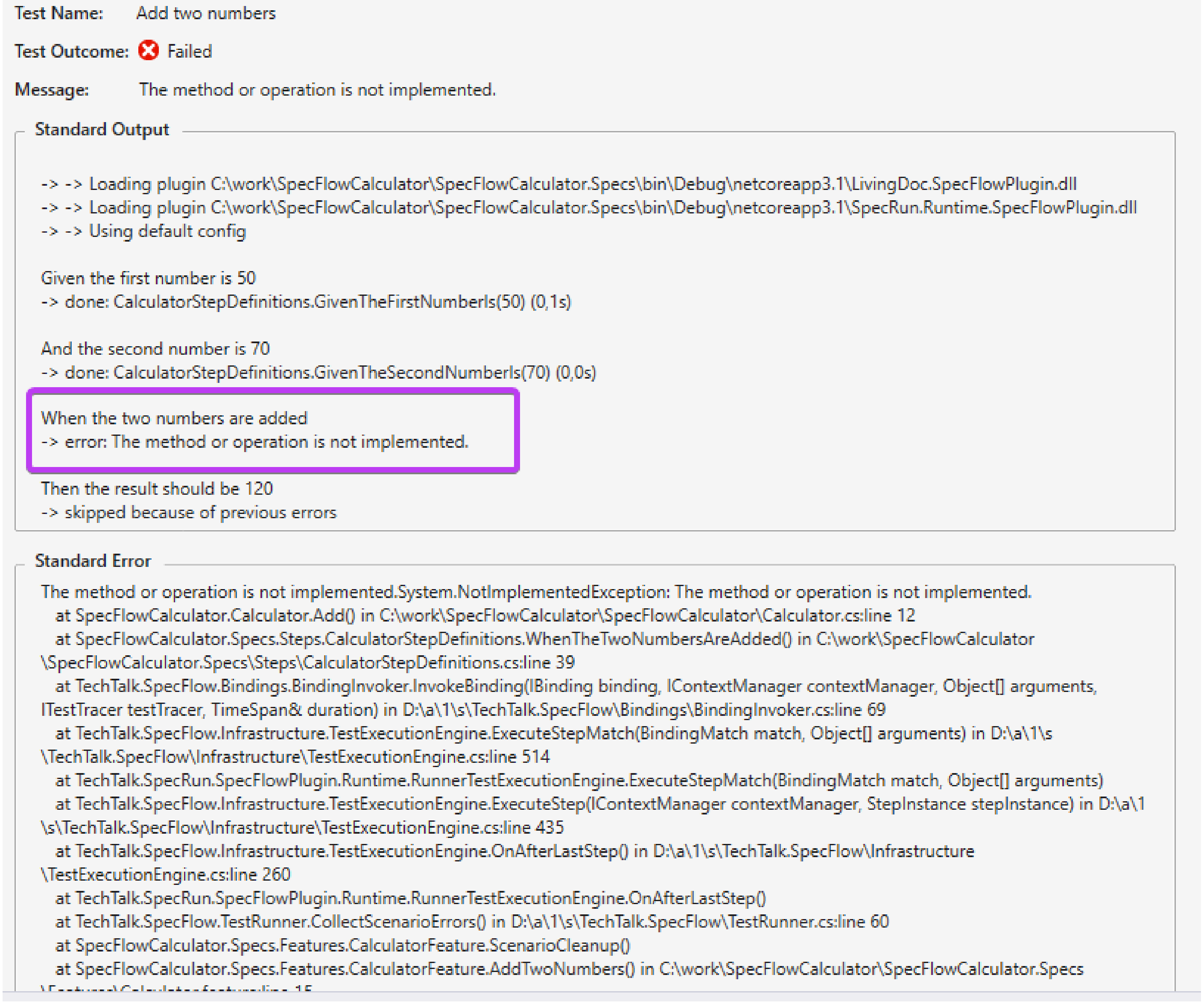
Now that we learned how to leverage BDD, our end product will be a software that matches the requirements and set of automated tests that checks the system against specifications, we need to execute these tests.

Luckily, cucumber integrates very well with famous test frameworks for all languages.

E.g., for c#/dotnet you'd use `dotnet test` command to run the tests as you'd normally do for unit tests written in `XUnit or NUnit or MSTest` . The same apply to all the other frameworks.

With that said, you can add the tests easily to your CI pipeline to ensure that every build respects the specifications of the system.

If error occurs from running BDD tests, the output is usually very verbose, which helps you to detect at which step exactly the error has occurred. This isn't the case if you write tests in code only (e.g., via XUnit as you'll get a stake trace of the error with no link or relation to the business feature). Check the example below:



```
Test Name:      Add two numbers
Test Outcome:   ✗ Failed
Message:        The method or operation is not implemented.

Standard Output

  -> -> Loading plugin C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\LivingDoc.SpecFlowPlugin.dll
  -> -> Loading plugin C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\SpecRun.Runtime.SpecFlowPlugin.dll
  -> -> Using default config

  Given the first number is 50
  -> done: CalculatorStepDefinitions.GivenTheFirstNumberIs(50) (0,1s)

  And the second number is 70
  -> done: CalculatorStepDefinitions.GivenTheSecondNumberIs(70) (0,0s)

  When the two numbers are added
  -> error: The method or operation is not implemented.

  Then the result should be 120
  -> skipped because of previous errors

Standard Error
  The method or operation is not implemented.System.NotImplementedException: The method or operation is not implemented.
     at SpecFlowCalculator.Calculator.Add() in C:\work\SpecFlowCalculator\SpecFlowCalculator\Calculator.cs:line 12
     at SpecFlowCalculator.Specs.Steps.CalculatorStepDefinitions.WhenTheTwoNumbersAreAdded() in C:\work\SpecFlowCalculator
  \SpecFlowCalculator.Specs\Steps\CalculatorStepDefinitions.cs:line 39
     at TechTalk.SpecFlow.Bindings.BindingInvoker.InvokeBinding(IBinding binding, IContextManager contextManager, Object[] arguments,
  ITestTracer testTracer, TimeSpan& duration) in D:\a\1\s\TechTalk.SpecFlow\Bindings\BindingInvoker.cs:line 69
     at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.ExecuteStepMatch(BindingMatch match, Object[] arguments) in D:\a\1\s
  \TechTalk.SpecFlow\Infrastructure\TestExecutionEngine.cs:line 514
     at TechTalk.SpecRun.SpecFlowPlugin.Runtime.RunnerTestExecutionEngine.ExecuteStepMatch(BindingMatch match, Object[] arguments)
     at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.ExecuteStep(IContextManager contextManager, StepInstance stepInstance) in D:\a\1
  \s\TechTalk.SpecFlow\Infrastructure\TestExecutionEngine.cs:line 435
     at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.OnAfterLastStep() in D:\a\1\s\TechTalk.SpecFlow\Infrastructure
  \TestExecutionEngine.cs:line 260
     at TechTalk.SpecRun.SpecFlowPlugin.Runtime.RunnerTestExecutionEngine.OnAfterLastStep()
     at TechTalk.SpecFlow.TestRunner.CollectScenarioErrors() in D:\a\1\s\TechTalk.SpecFlow\TestRunner.cs:line 60
     at SpecFlowCalculator.Specs.Features.CalculatorFeature.ScenarioCleanup()
     at SpecFlowCalculator.Specs.Features.CalculatorFeature.AddTwoNumbers() in C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs
  \Features\Calculator.feature:line 15
```

Documentation

While that's great! Still the stakeholders, end-users, and business teams won't have access to our version control to read the gherkin files that're stored alongside the code, even if they have access it'll be hard for them to navigate the source control to find what they're looking for.

There are a lot of tools to address this problem, by integrating such tools to the CI, you can generate documentation straight from the gherkin files. An example to such tools is `SpecFlow LivingDoc` which given a gherkin files, generates an HTML documentation website with no dependencies to enable you to share it as you wish.

Conclusion

We demonstrated how BDD is a powerful tool that enhances the day-to-day work of both business & technical teams, as within its heart focuses on alignment and eliminates any misunderstandings by adhering to a set of standard and common processes that are easy to use for technical and non-technical users alike.

We also learned that BDD results in concrete product specifications written in a ubiquitous language living alongside the system code, bound to it, and validates the system against it with every CI run.

Moreover, we learned how the gherkin specifications we wrote could be a single source of truth for the system as it may hold the specifications, which we use to write test automation and generate documentation for the systems’ users and non-technical people.

Disclaimer: I certify that generative AI, incl. ChatGPT, has not been used to write this essay. Using generative AI without permission is considered academic misconduct

References

[1] https://www.businesswire.com/news/home/20210216005484/en/Rollbar-Research-Shows-That-Traditional-Error-Monitoring-Is-Missing-the-Mark

[2] https://web.archive.org/web/20150901151029/http://behaviourdriven.org/

[3] https://cucumber.io/docs/bdd/

[4] https://www.geeksforgeeks.org/acceptance-testing-software-testing/

[5] https://www.atlassian.com/agile/project-management/user-stories#:~:text=software user’s perspective.-,A user story is an informal%2C general explanation of a,value back to the customer.

[6] https://specflow.org/