DRAFT

How Behavior Driven Development together with Cucumber can improve your development process

by DXC Technology

September 2018

Created by:
Manuel Wetzel

Abstract

This white paper outlines what Behavior Driven Development is. Therefore Domain Driven Design and Test Driven Development are explained and all three are compared and the advantages are analyzed.
It is also pointed out in which case BDD makes sense and when it is more of a hindrance. This white paper also contains a concrete business case to show which problems could be solved through BDD.
It is also explained how BDD can be used together with the software tool Cucumber. Therefore a small example is given which shows how the development process works.

1 What is BDD

Behavior Driven Development is a software methodology invented by Dan North. He was unhappy with agile practices like Test Driven Development and started to use them differently. There are many slightly different definitions for BDD. But all describe it as a combination of Domain Driven Design and Test Driven Development.

1.1 Domain Driven Design

When a team is working on a software project it can happen very fast that they lose the overview over the whole project. They have no distinguishable architecture and code that works, but without explaining why. This happens because domain complexity is mixed with technical complexities. Domain Driven Design tries to solve this problem.
Complex applications are made less complex by connecting related pieces of software into an ever-evolving model. This is achieved by focusing on the core domain and the domain logic. Which requires a constant collaboration between developers and domain experts. Together they try to split complex designs into smaller models of the domain. To ensure that the code reflects the domain.
Most often domain experts can't read code and the developers have just as little knowledge of the domain. So they need a ubiquitous language to communicate successfully. If they have an idea which can't be easily expressed with their common language then they know that there is something missing from the domain model and they have to work together to find that missing part.

1.2 Test Driven Development

Test Driven Development is a software development methodology where tests are written before any actual code is written. There are five core steps which are repeated again and again throughout the software development lifecycle:

1. Write the smallest possible test that is necessary to check the feature that you want to add. You don't need to cover everything that can go wrong with that feature, because your feature test is no replacement for unit tests. You will still need to write unit tests to ensure that your feature is save in every aspect.
2. Run your test and check if it fails. This is a key aspect of TDD because you ensure that your test

covers the right thing and that the desired feature isn't already implemented.
3. Write code to get your test pass. Here it is important that you don't write too much code because that is what causes bugs later on.
4. Run your test again and check if it passes. A passing test is a sign of a working feature.
5. Refactor your code. Take a step back and check if your implementation isn't interfering with other code. It is also important to run the tests of previously integrated features, to check if your new feature broke anything.

There are some obvious advantages in the usage of TDD like having tests at all or no unnecessary code. But there are a lot more. TDD will keep you focused on your goal because you have an executable checklist of all the features you have to implement. It will help you think about the key features that your software really needs.
The most important part is that you will work better with your team. You can explain the functionality of your code pretty easy to other developers just by showing them your tests.

1.3 Behavior Driven Development

Behavior-Driven Development (BDD) is a set of practices that aim to reduce some common wasteful activities in software development:

- Rework caused by misunderstood or vague requirements
- Technical debt caused by reluctance to refactor code
- Slow feedback cycles caused by silos and hand-overs

BDD aims to narrow the communication gaps between team members, foster better understanding of the customer and promote continuous communication with real-world examples.
Examples describe how the software is intended to behave, often illustrating a particular business rule or requirement.

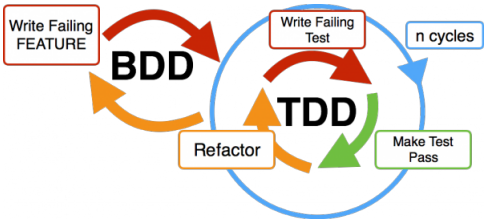
1.3.1 Specification Meeting

The first part is a meeting where the people in charge of defining the requirements (business analysts) sit down with programmers and testers to discuss a feature to be implemented.
This might seem like a waste of time to involve technical people in the specification of the requirements but they can give valuable input regarding the implementation of the requirements.
The result of this meeting will ideally be multiple steps written in Gherkin. Gherkin is a language with a loose syntax where features can be described with English sentences. But even if the steps aren't written in Gherkin in the meeting, the developers and testers got a clear understanding of how the feature should behave and it will be easy for them to write that down in Gherkin syntax.
The specification meeting will encourage discussions about details of the feature which will lead to a better product before programming has even started.

1.3.2 Outside-In Development

The second key element of BDD regards the programming.
Outside-in development describes a process where programmers start developing an application with the functionality that is closest to the user and gradually work towards the guts of the system (business logic, persistence...) as they discover more of what needs to be implemented.
With a tool like Cucumber, this means that they start by executing their feature written with Gherkin syntax. Cucumber then tells them which tests to implement. There will be a test for every step of every scenario.
The developers execute their project again and get a lot of failing tests because they have written tests for something they haven't implemented yet. They implement the code that is needed to make the tests pass and if no failing steps are left, the specified feature is implemented with the behavior specified in the meeting with the business analysts.

The process in BDD where you write tests before you start with actual features is the same as in Test Driven Development. The main difference is that BDD operates on a higher abstraction level, closer to the domain and farther from classes and methods. This ensures that non-technical stakeholders can contribute and review at any time of the development cycle.



2 What makes BDD Different

Experienced developers discovered, that TDD was not primarily about the tests, but about the behavior. Behavior Driven Development aims to fix the terminology first, by referring to behavior instead of test. This is combined with aspects from Domain Driven Design to close the gap between Business users and Technologists. It's all about getting the words right.

2.1 Advantages

Since Behavior Driven Development is a combination of TDD and DDD it has many of the advantages from them.

2.1.1 From TDD Perspective

Developers need less debugging because if they write the tests first they need a deep understanding of logic and functional requirements before they even start with the actual code.
The user experience is in the focus. Because developers have to think backwards when they write their tests first, they get encouraged to think about the user experience of their new feature. This increases the overall quality of the product.

Overall Development time can be decreased. Although the lines of code increase due to the tests, they can save a lot of time. New bugs will be caught earlier through the suite of tests that is being executed continually.

2.1.2 From DDD Perspective

The communication is easier because everyone speaks the same language. The ubiquitous language will define simpler terms to refer to the more technical aspects. This means that even non-developers can read the tests because they are named in the ubiquitous language.
The flexibility is improved because everything is based on the domain model. This means that the implementation of various components can change since only the expected behavior is specified. The focus is more on the domain than on the interface.

2.1.3 New Advantages

Different from TDD, in BDD you don't change the design or add a new class or function if there isn't a change in the product. This decreases the chance that a unnecessary change of the codebase is made, which could cause bugs.
You don't need to test individual classes or functions. In BDD it's all about the features and that is all you need to test from the BDD perspective. Although this doesn't mean that additional Unit tests are a bad idea.

2.2 Possible Problems

Behavior Driven Development is no silver bullet and there can occur a lot of problems. The first problem comes with one of the biggest advantages of BDD. The communication between developers and the business owner. In a team of a small agile company which develops software for themselves, this might be no problem. But imagine a larger organization who's developing a solution for a customer. The customer is the domain expert and he needs to set the requirements. In a BDD style, the developers would need to have constant contact with the domain expert from outside the company which might be very difficult if he works on other projects at the same time and isn't always available for instant feedback. One way to avoid this would be proper resource planning. The right amount of foresight will minimize the demands on the business owner.
Another problem is that BDD doesn't work if there are still legacy silos in the team. If a team isn't fully agile with flexible roles the collaboration will get very difficult. And BDD will be useless if business people don't contribute and collaborate in defining, creating and maintaining the tests.
A problem that affects larger organizations, is that BDD requires scripting skills. Not everyone, but at least testers should be able to script automated tests. If a lot of testing is done manually the decision to teach every tester to code or script might be a bigger step for the organization.
The requirement to write a detailed specification of each feature together as a team can be a problem too. For larger teams who work on larger projects this might be no problem, but for small teams who work on rapidly changing projects, the regular meetings can be more hindrance than help. Another important

thing is to refactor. Not only the code but also the tests. During behavior driven development the code gets refactored every time you add a new feature. And it will happen that you need to move tests because the behavior moved elsewhere or that you have to delete tests because they are no longer up to date. If you don't do that the test suites become unwieldy, the test execution takes a long time and you get bugs in code you don't need.

3 When is BDD Useful

Behavior Driven Development is useful when a team is creating software. For one developer alone it adds no value because he knows his product and he can read the code if he forgot anything.
In a software development team though there are many developers working together. Of course, they all can read code, but it can be really difficult even for an experienced programmer to figure out what the code, written by someone else, does. In addition to the developers, there are other roles in a larger development team. Stakeholders like testers have only limited knowledge of code and collaborators like the project manager and most important the business owner most often can't read code. But the business owner is one of the most important roles in a team because he is the one responsible for the right outcome. He represents the demand of the customer. If the business owner translates the requirements wrong, then his team ends up building software no one has ever asked for. And if they are developing in a non-agile way, the whole team worked maybe half a year or even a whole year for nothing.

Another scenario would be if separate teams are working on the same project where one is responsible for the development and the other team for the quality assurance. When working in different teams unclear requirements are becoming a big problem very fast. But even if the requirements are well defined each team develops its own ubiquitous language which makes communication ineffective.
More communication problems occur if the business owner defines the requirements without other team members. This makes it very hard for the developers to understand them.

4 BDD Applied Wrong

Because Behavior Driven Development a methodology and there is not only one way of doing it right. BDD can be adapted to the needs of your team.
But there are some things that caused problems in previous projects.
As Monica Obogeanu describes on mozaicworks.com, one mistake she made with her team was to skip the conversation. They left out the meeting where product owner, developer, and tester should define the specifications for the project. The outcome was, that scenarios were missing and time was wasted with treating the missing scenarios.
Another mistake was to keep the outcome of the specification meeting as one. This makes the development more complicated and leads to more complex test suits. It's easier to split the specification into smaller parts.
The last thing Obogeanu mentions is that you shouldn't expect that everything will be correct right from the beginning. When problems occur it is important to collaborate quickly to resolve them.

Another thing that can cause problems when using BDD is when no unit tests are written. Behavior Driven Development is about the behavior and that is being tested. But these tests are no replacement for unit tests which are faster and ensure code coverage.
According to John Ferguson Smart, something that he experienced as an obstacle for teams starting with BDD was that they started immediately using certain tools like Cucumber and focused too much on writing the specifications in Gherkin syntax. This slows down teams extremely and the specification meetings are getting way longer. There are many ways in which the specifications can be written down in the meeting. It would be faster if one developer translates the specifications into features in Gherkin syntax and hands them over to the business owner to check them. Features in Gherkin syntax are easy to read but difficult to write.

5 Concrete Business Case

5.1 Initial Situation

Imagine a banking company with several small teams of around 10 people each. There are many roles in this team like developers, testers, business analysts, domain experts, someone from operations and a business owner. These teams work together on a large software product that plays an important role for the company. A huge number of changes is being delivered with each release and just a little bug can easily cost billions of dollars. But the banking company relies heavily on software and so the development teams are very careful in what they produce and what they actually release to the live system.
All the requirements are written down in an issue tracker in natural language. This causes misunderstandings between the product owner and the developers which lead to software that no one has asked for. Additionally, the testers use their own tools and work isolated from the developers with their own understanding of business rules and requirements.
The development process runs in 3 independent phases. In the first phase, someone from the business works with the customer to define the requirements and phrases them for the developers. In the second phase the developers code the application and in the third phase the testers test the code.

Large companies also often have huge issues with trust. Mistrust between business stakeholders and developers comes from the fact that business stakeholders have no insight into the development process because they can't read code. So they have to wait until changes are implemented to validate them. Mistrust between team members also evolves through the fact that developers deliver buggy code despite writing unit tests or testers writing large, manual test suites that take a long time to run.

5.2 Improvements through BDD

If the banking company starts using BDD the first improvement would be that all the team members would work together instead of working in different phases. This would shorten the development time tremendously because there would be no back and forth with every misunderstanding. This collaboration would establish a shared understanding of the application and the same tools would be used. This understanding would be recorded in the form of executable specifications in such a shared tool. These

specifications could be used from the testers to write tests for the expected behavior and developers would have the tests during the development process to ensure that they are implementing the right features that are needed. Because the specification is written in English sentences with a loose syntax, everyone, including the product owner, could read them and see how far the development process is and if the application behaves as expected. This would eliminate the trust issue.
The last benefit would be that the developers and testers would have more confidence in what they release because they would know that no unnecessary code is included in the code base that could cause bugs and through all the behavioral tests they would know that the application behaves as expected.

6 BDD in Practice with Cucumber

To use BDD efficient a tool is needed. Cucumber is one of many. Cucumber has good support and is available for many languages like Java, Javascript, Ruby, C++ and many more.

6.1 What is Cucumber

Cucumber is a tool that supports Behavior-Driven Development (BDD).
Cucumber reads executable specifications written in plain text and validates that the software does what those specifications say. The specifications consist of multiple examples, or scenarios, like:

```
Scenario: Withdraw money
    Given i have $300 in my account
    When i withdraw $100
    Then my balance should be $200
```

Each scenario is a list of steps for Cucumber to work through. Cucumber verifies that the software conforms to the specification and generates a report indicating success or failure for each scenario.

In order for Cucumber to understand the scenarios, they must follow some basic syntax rules, called Gherkin.

6.2 A small Example

Let's say you want to develop software for an atm.
So a developer, a tester and someone who knows the domain, like a business analyst, a product owner or a customer sit together in the first meeting. They start by describing the first feature which is the withdrawal. The feature contains several steps and is written with Gherkin syntax and might look like:

```
Feature: Withdraw money
        Customers should be able to withdraw money from the atm.

   Scenario: Withdraw money
      Given i have $300 in my account
```

```
    When i withdraw $100
    Then my balance should be $200

  Scenario: ...
```

The development process starts by executing the feature file. Cucumber will then tell you what to implement: (Java example)

```
You can implement missing steps with the snippets below:

@Given("^i have \\$(\\d+) in my account$")
public void i_have_$_in_my_account(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^i withdraw \\$(\\d+)$")
public void i_withdraw_$(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^my balance should be \\$(\\d+)$")
public void my_balance_should_be_$(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}
```

After the Given, When, Then annotations are regular expressions which map each step from the feature file to a class which is called step definition. The developer (or tester) has to implement tests of what is described in the class name. If he changes nothing in a class a pending exception is thrown which indicates that there needs to be something done in the future. The step definition file might look like:

```
public class StepDefinitions() {
    @Given("^i have \\$(\\d+) in my account$")
    public void i_have_$_in_my_account(int balance) {
        atm.setBalance(balance);
    }

    @When("^i withdraw \\$(\\d+)$")
    public void i_withdraw_$(int amount) {
        atm.withdrawMoney(amount);
    }

    @Then("^my balance should be \\$(\\d+)$")
```

```
    public void my_balance_should_be_$(int balance) {
        assertEquals(balance, atm.getBalance());
    }
}
```

After implementing these lines of code the developer runs the tests to see if they all fail. This is important because if the test would pass this would be a sign, that the expected behavior is already implemented or that the test is testing the wrong thing.
The next step would be to actually implement the functions of the ATM so that the tests pass.

If every test passes the software behaves as in the feature file specified and you start from the beginning with your next feature.

7 Conclusion

Behavior Driven Development is a practice that can help your team build software. It can improve communication with the specification meetings where testers, developers, and business analysts collaborate. It can also improve code quality due to the test first practice. This increases the code coverage and ensures that the features are implemented right. To benefit from these advantages BDD has to be applied the right way. Else there can occur several problems that slow down the development process.
Behavior Driven Development is useful for teams that are ready for a change and that are willing to try out something new. For large teams in large organizations, this can be a problem because of the lack of flexibility. This could make it really hard to start using BDD.
BDD is only a practice without any restrictions on how to use it. This is an advantage because every team can adapt it to their needs. There are a lot of tools which help when using BDD.
One of them is Cucumber. When working with Cucumber the features have to be specified with a loose syntax. Cucumber is able to execute these files and to provide a skeleton where the tests can be implemented very easy.