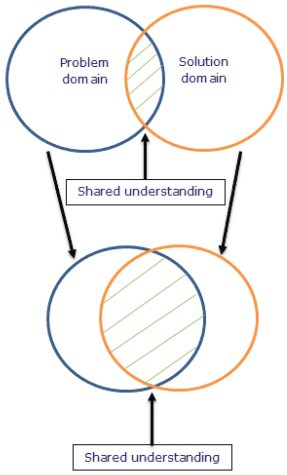
Behaviour Driven Development (BDD) in Cucumber is the ultimate solution. BDD in Cucumber won't get the job done. Or is there a more nuanced approach? On September 22 Matt Wynne, author of "The Cucumber Book", visited Eindhoven to tell about the added value and the limitations of BDD. This is my version of his story with some editorial notes.

An age-old problem that software development faces, is the gap between requirements and implementation. The client asks for a portal, but actually means he wants an intranet. The software development team interprets a portal as a portal. Lo and behold, the first misunderstanding.

It's true that we can come a long way if we apply the Agile principles correctly. But more often than not, many (Agile) projects become too technical, causing shared understanding to decrease.

BDD aims to increase shared understanding about the desired behaviour of software. In other words, correct application of BDD brings the problem domain (business) and the solution domain (software development) closer to one another.



Cucumber

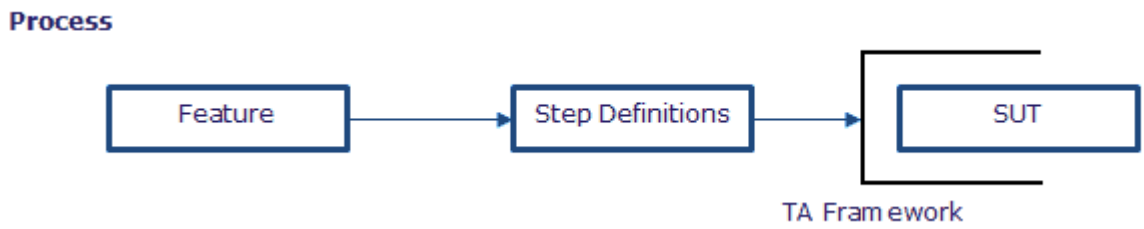
Cucumber is a collaboration tool which facilitates in forming a shared understanding of the desired behaviour of software. Cucumber executes specifications which are written in natural language. These specifications are called features. Besides being specifications, they function as automated acceptance tests and (living) documentation. This makes BDD a Test-First approach.

Features and their respective scenarios ought to be written by the three amigos. These are: the product owner/business analyst, the tester and the developer. Their conversation about features and scenarios increase shared understanding and help reveal “unknown unknowns”, better known as contingent obscurities.

Gherkin is the syntax for writing features. An example of a feature written in Gherkin can be seen below.

Feature: Login
Scenario: Successful Login
Given I am not logged in
When I login with user “test” and password “test”
Then I should be logged in and see my profile

A team should consider setting standards for the desirable level of detail in features. Another important aspect is clarifying semantics in order to form a ubiquitous language. But it would be too much of a digression to discuss these aspects in this article.



BDD is also known as Specification By Example, because a feature is an executable specification. Nothing actually happens when a feature is executed. Step definitions are required to make a feature do something. An example of the When-step in Ruby:

```
When(/^I login with user “(.*?)” and password “(.*?)”$/) do |username, password|
@username = username
@password = password
…
end
```

After writing step definitions and executing the feature, tests fail. Failing tests trigger us to write the necessary code in the test automation framework and/or the application (system under test). But it’s not over when tests pass. Refactoring still needs to be done. Refactoring is changing the code’s structure without changing its behaviour. Refactoring aims to increase reusability, readability and efficiency. The feature needs to be executed again after refactoring. This is to make sure the software still displays the desired behaviour.

Life after BDD

“We all know that silver bullets are bullshit”, with this statement Matt Wynne started his discourse about life after BDD. Applying BDD can be advantageous. Yet, if one sees it as the ultimate solution, the disadvantages outweigh the advantages. A software development team needs to realise there is life after BDD. Matt explained this by talking us through the six phases software development teams (usually) go through as they learn to apply BDD.

Phase one is called: “Let’s make toast the American way! I’ll burn the toast, you scrape it”. First developers make software. Then testers manually execute tests and find bugs. A project manager prioritizes bugs, developers fix them and the story repeats itself starting with testers finding bugs.

Phase two is called: “I’ll burn the toast, you’ll automatically scrape it”. First developers make software. Then testers write automated tests, execute those tests and find bugs. A project manager prioritizes bugs, developers fix them and the story repeats itself starting with testers finding bugs.

This is not BDD, even if a team were to use Cucumber. The team is building software backwards because verification of correct behaviour (acceptance criteria) is still an afterthought.

Phase three is called: “Three amigos”. First, the three amigos have conversations. These result in understandable features and their respective scenarios. Developers write code to make features pass. Testers automate (different) tests. Testers find bugs during exploratory

testing. A project manager prioritizes bugs, developers fix them and the story repeats itself starting with testers finding bugs.

Phase four is called: “Test first”. First, the three amigos have conversations. These result in understandable features and their respective scenarios and clear, abstract rules. Developers automate the step definitions and test help them in finding the edge cases. Bugs are now called new scenarios. So testers find new scenarios during exploratory testing. A project manager prioritizes the new scenarios, and the story repeats itself starting with adding scenarios.

Phase five is called: “Disillusionment”. Builds take forever and fail frequently. There are flickering scenarios which fail arbitrarily. The amount of features has increased dramatically and there are way too many slow tests on UI level. This is also known as the ice cream cone anti-pattern. The team now shares an enemy: Cucumber.

Many teams give up at this point. They put the blame on Cucumber. But it’s not Cucumber’s fault. When teams go through phase five, it’s time they examine their features. Does a given scenario still have added value? If not, delete it. If so, can the test be pushed down the test level stack? If it’s possible to turn it into a unit (integration) test do it and delete the scenario.

Phase six is called: “Transcendence”. The whole team has a shared understanding of the problem domain and is able to make a fitting solution. Scenarios are readably and understandable for everyone. The emphasis is no longer on the amount of scenarios, but the abstract rules embedded in those scenarios. There is a sound test (automation) strategy which determines on which level test have to be specified.

Conclusion

BDD is a useful practice when it’s combined with other good practices. BDD is no silver bullet, but a way to come to a shared understanding. Cucumber facilitates in forming that understanding. BDD is hot and Cucumber is useful with
in these boundaries.

David Baak, test specialist at Polteq