



# Ebook

## Kick start and scale BDD across your organization



# Table of content

## Why this ebook?

1. Adopting BDD: a cultural shift
2. Let's have the conversation
3. Let's automate the checks
4. Living documentation
5. BDD at scale: a ticketing system for the city of Helsinki

# Why this e-book?

Behavior Driven Development (BDD) is an approach that consists on defining the behavior of a feature through examples in plain text. These examples are defined before the development starts and are used as acceptance criteria. They are part of the definition of done. That's a very powerful approach that we promote and use at Hiptest in our development process.

These examples support the conversation and help the cross functional team (marketing, product owner, developer, user...) to create a shared understanding of what should be developed. This approach to development has proven to be a game changer amongst agile and DevOps teams. That's a great way to minimize waste and avoid development of features that nobody wants or that do not meet the business expectations.

These concepts are quite simple. But implementing them is much harder. With this e-book we want to give back to the community and share the good practices we've learnt the hard way.

# **Adopting BDD: a cultural shift**



**In the past decades, software development lifecycle has been structured with silos. Every silo comes with its repository.**

Agile development has changed the game and broken the silos. There is also a lot of buzz around agile testing. But BDD is more than just testing. Trying to match with the concepts of the old world is useless. It is like applying equations of the classical physics to the quantique world. When doing BDD, specifications and tests

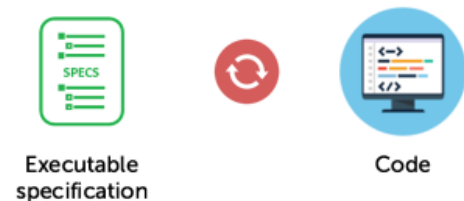
### From Waterfall...



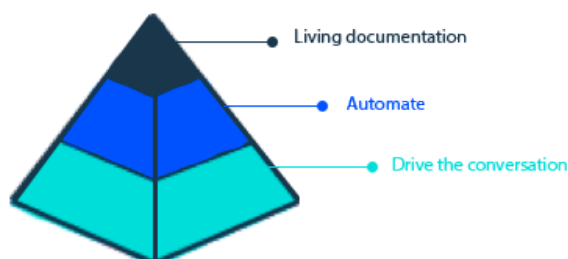
become two sides of the same coin. You define your specification by examples and at the same time you get your tests. These specifications become executable when the examples are automated.

And this new paradigm changes everything. You'll face many questions further down the road towards BDD. If you want to go deeper on the theory and cultural shift, I suggest the following reading: "Specification by Example" by Gojko Adzic

### ...to BDD



Having now more than 1000 teams using BDD in our community, we've been able to define 3 different levels of maturity:



product and development team.

1. BDD used as an opportunity to have the conversation between all the stakeholders (developers, testers, Ops, BA, PO, marketing ...)
2. Using the examples and automating them as part of the non-regression testing activity
3. Using the automated scenarios as living documentation for the marketing,

We detail these three different stages in the following part and share an example of BDD used at scale.

**Let's have the conversation**

2.

### a. Start with the benefits in mind when describing the feature

When you have a new feature to develop, you should start the discussion by focusing on the benefits of this feature. In this conversation, each role should be represented: business experts, developers, testers.... These participants should answer the WHY: why should we develop this feature? That's a great way to drive the conversation on what really matters. In BDD, a feature has a description that provides the context. It is usually written this way:

***In order to*** [get a benefit]

***As a*** [role]

***I want*** [a feature]

Let's make an example. At Hiptest, we decided to develop an integration with Slack (the real time messaging App) a couple of months back. It was requested by many customers to get updates about the testing project directly from slack. The development team quickly identified two use cases that led to two different authentication architectures. So impact on the design was important. So, before investing a penny on development, we discussed the two options with customers and chose the option that had the biggest impact. Having this discussion before starting development was really time saving.

Now that we have a clear understanding of the benefits of the feature, let's write our scenarios.

### b. Capturing the conversation with scenarios

With your scenarios, you'll be able to describe the use cases through examples. At this point, focus on the WHAT not the HOW! To write your scenarios we usually recommend to use the Gherkin syntax.

Gherkin is plain-text language with a little extra structure designed to be easy to learn by non-programmers. It allows concise description of examples to illustrate business rules in most real-world domains. One of the big advantages is that it clearly highlights the intent of the example/test.

Follow these basic steps to start creating your scenarios using Gherkin syntax:

- As a team, go through your user stories and write BDD scenarios using the keywords GIVEN, WHEN, and THEN (AND, BUT can be used as well)
- GIVEN is your setup; for example, "GIVEN the credit card is valid"
- WHEN is your action; for example, "WHEN I request \$50"
- THEN is your assertion; for example, "THEN the ATM should dispense \$50"
- Capture the BDD scenarios in a location that is public for everyone to see. Hiptest is a good choice 😊

Here is a sample Gherkin document:

**Feature:** Account Holder withdraws cash

**Scenario:** Account has sufficient funds

Given the account balance is \$100

And the card is valid

And the machine contains enough money

When the Account Holder requests \$20

Then the ATM should dispense \$20

And the account balance should be \$80

And the card should be returned

Once the team is clear on the feature to be developed thanks to the scenarios, developers can start the implementation. They will be driven by these acceptance examples. It makes the process more efficient.

Considering technical feedbacks, examples can be updated with the stakeholders during the development. After development, these scenarios can be executed manually or automatically to check that the business needs are met.

### c. Good practices to write your scenarios

They are 2 ways to write your examples: imperative VS declarative style. With imperative style, scenarios tend to be long with very low level steps that drive the user interface. When using the declarative style of Gherkin syntax, scenarios describes the what, not the how. We strongly recommend the declarative style as they are focused on the business rules.

Ex: *Given I am logged in*









We don't care about how we did the login (with a username and password or touchId Facebook login...). The thing that matters from a business stand point is that *I'am logged in*! Detail of the login procedure will be pushed into the step definitions when automating the scenario.

You'll see that this approach of writing your scenarios will make them shorter and using a consistent business terminology.



You should also make sure that this business terminology is used consistently across all your scenarios. This way it becomes the common language shared by all the team. You'll need auto-completion to ensure consistency and refactoring capabilities to manage the impacts of changes.

Our last suggestion will be to review and refactor your scenarios continuously. This will increase test readability and so facilitate maintenance. These are good practices for your code and the business users should keep the same good habits while writing scenarios. When modifying a step, all the scenarios using it should be impacted automatically.

1	Given I have activated Slack in the settings of my Hiptest project Coffee Machine
2	When I query /hiptest TestRunID from slack
3	Then I should see the breakdown of tests by statuses for TestRunID
and I should	
 Create action word	
 "user" should be logged in (called 2 times)	
 I should get a notification "p1" (called 8 times)	
 I should be member of project "p1" (called once)	
 the file should contain an Index Tab (called once)	
 I should have a file in the download folder (called once)	
 used by section should contain "p1"	
 ...	

If you stop there you'll learn a lot about the feature to be developed and will be able to focus your development efforts on what really matters.

Of course it is great to automate the scenarios and get automated checks. You'll also unleash the value of living documentation that is really amazing when you get to that point.

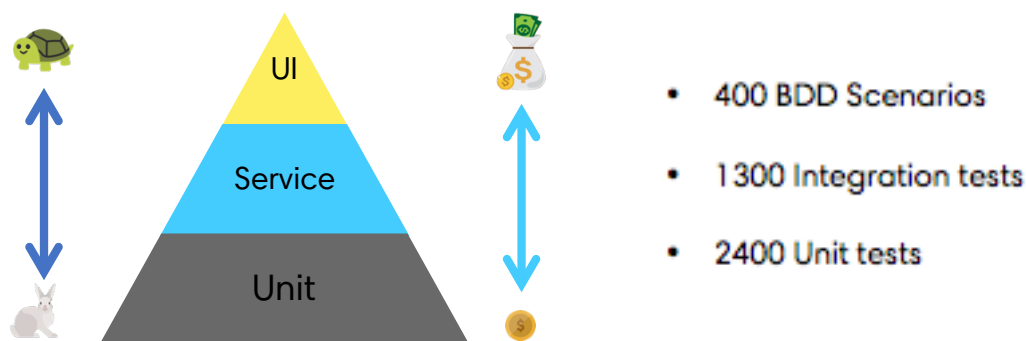
**Let's automate the checks**

3.

The most common bias we see is “I want to automate my tests using BDD”. So, let’s make it clear: BDD is not about test automation. It will not ensure the non-regression of your software. It contributes to it and provides some checks on either the business expectations are met or not.

#### a. BDD VS TDD

To ensure the non-regression of your product, you mainly rely on the unit tests and integration tests. As an example this is how the testing pyramid looks like for Hiptest platform:



Test-driven development (TDD) is a style of programming where coding, testing, and design are tightly interwoven. Benefits include reduction in defect rates and the availability to develop step by step small increments. This creates a unit tests safety net which insures security during the refactoring. TDD consists of the following steps:

1. Start by writing a test
2. Run the test and any other tests. At this point, your newly added test should fail. If it doesn't fail here, it might not be testing the right thing and thus has a bug in it.
3. Write the minimum amount of code required to make the test pass
4. Run the tests to check the new test passes
5. Optionally refactor your code
6. Repeat from 1

Behaviour Driven Development is an extension of Test Driven Development that involves business stakeholders. Units of code (individual methods) may be too granular to represent the behaviour represented by the behavioral tests. That's why you need both to test a feature appropriately.

#### b. A test automation framework that scales

There are good practices that can help you when building a test automation process that scales.

## **#1 Test automation is not about ROI but about velocity**

If you think of automation as no more manual work to do, you'll be disappointed. Using test automation is expensive and requires a lot of work and maintenance. And the reason why we may use it and have a pay back is velocity. Automation is about speed of feedback. It is not about cost and it is an illusion that automation means \$0.

Parkeon, one of our customer and leader in the smart cities industry, has developed a huge system (more than 300 man/years) to manage ticketing and transportation for a big city. They have adopted the agile practices and built thousands of automated tests using BDD. Part of the tests were executed at the API level, part at the GUI level and a few of them on real devices with a robot to give the input.

As you can imagine, building such an infrastructure was expensive. But it was worth the investment because the value they were looking for was speed of feedback. They were able to detect unanticipated impacts due to changes in the code very quickly. That enables the team to iterate faster, add new or modify features with confidence.

## **#2 Be careful with record and playback tools**

You use test automation and try to scale your practice with record & playback tools? Then you push the snow in front of you and the more you move forward the bigger the pile of snow is. At some point you will not be able to move anymore. That is the maintenance problem all teams using record & playback face.

There is no magic and the promise "you can automate without any technical skills" is a dead end. Record and playback can be useful as a quick start strategy but then you should definitely modify and maintain the scripts directly at the code level.

## **#3 Make sure you have the skills**

As a consequence if you want to start using test automation, first make sure you have the right skills in your team. Basically you should only work on test automation if you have the ability to work on the production code, either you call these persons test automation engineers, developers or SDETs (Software developer engineer in test). And the code should live in the same repository as your application.

When you automate a test you have to decide if the behavior should be tested at the GUI level, API level or if there are redundancies with the unit tests. So there is a real discussion that has to happen between developers and testers. No collaboration with the developers means a really really bad starting point.

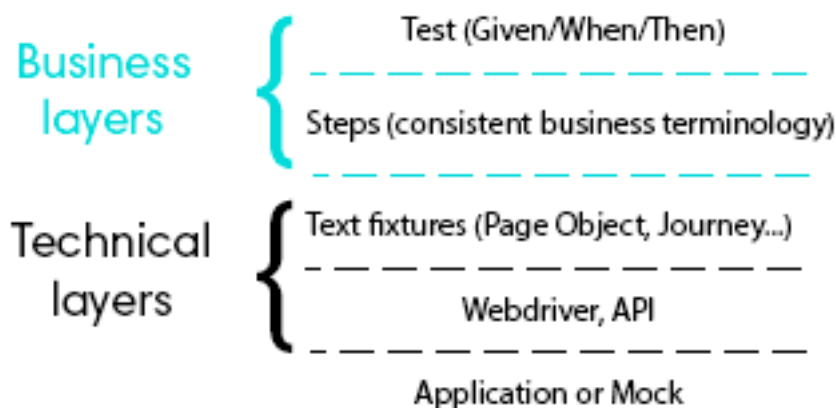
## #4 Every Test should tell a story about your application

One of the most popular question we get is: which tests should I automate? And our first answer is always the same: why have you created this script? It doesn't matter if the execution is manual or automated. Tell us what is the intent of this test! Guess what? Most of time the answer is not clear. So people are talking about automation, thus making a big investment and the "why" is not clear.

So make sure your tests are readable and with a clear intent. Every single test should tell a story about the application. This is where BDD really helps. Your tests/examples are both the living specification of your software and the acceptance tests for the developers: 2 sides of the same coin.

## #5 Build your test framework into layers

Last, if you want your test framework to scale then use a multiple layered approach.



At the business layer level you define your test using a consistent business terminology (steps). This is where auto-completion features and refactoring are really key when designing a test. Make sure you don't end up with "Sign in" step and "Login" step that have exactly the same meaning but written in 2 different ways. That will make the automation much more difficult.

Once you have your tests with a good design, readable and with a clear intent, it is time to implement the steps (test fixtures). There are many patterns you can follow for the implementation like Page Object or Journey pattern.

Having a framework with different layers will enable you to limit the impact when a change occurs and will make your maintenance much easier. That speeds up software delivery and improves quality and so, enables continuous deployment.

# Living Documentation

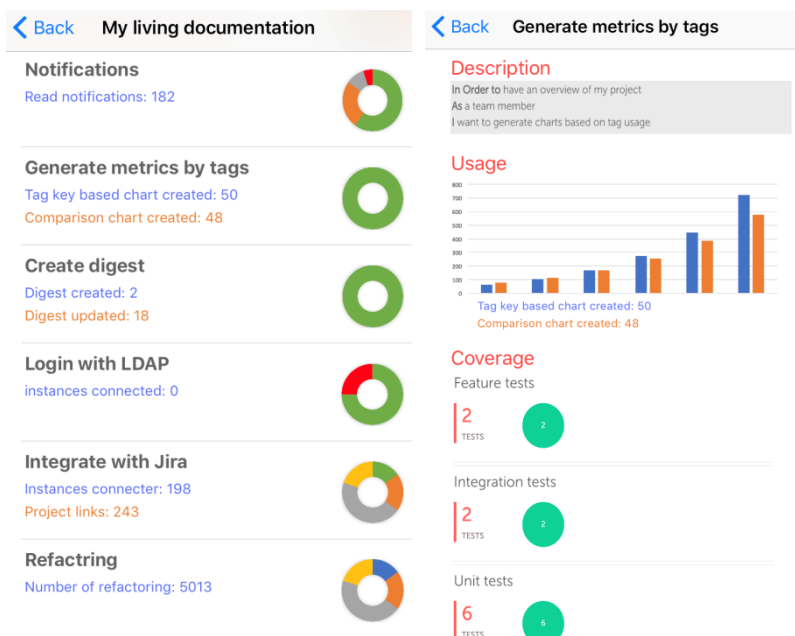
4.

When writing examples to describes the desired behaviour of the application and use them to drive the development, you write a documentation that all business stakeholders can review!

And once automated they become living documentation. Unlike a word document, the features and scenarios will always be up to date by design. These automated checks are executed as part of the CI process. The documentation reflects the true state of the system.

For example, the support team can search for the feature “Filter”, understand the behaviour and better answers to users...

And we have realized that the most successful team extend this approach by adding the context of the feature:



- Business assumptions related to the feature
- Current business metrics (usage, activation...)
- Performance

This way, every stakeholder, marketing, product owner or developer can look at a feature, understand the behaviour, get feedback from the production and measure the impact of the feature in real time! That's the value of living documentation. This is how we use it at Hiptest and it enables us to dramatically optimize our development effort. Which feature should we drop? which one should we invest on...? Business stakeholders and developers are using the exact same data to support discussion about the product and make decisions that will have the biggest impacts.

# **BDD at scale: a ticketing system for the city of Helsinki**





## Using BDD at scale

Among these cutting-edge companies using BDD, Parkeon has implemented this approach at scale on a very large project for the city of Helsinki: development of a ticketing system. This project required four years and hundred thousands of man days of development. The system is now in production and system's goal is to manage one million commuters per day and 6000 connected devices.

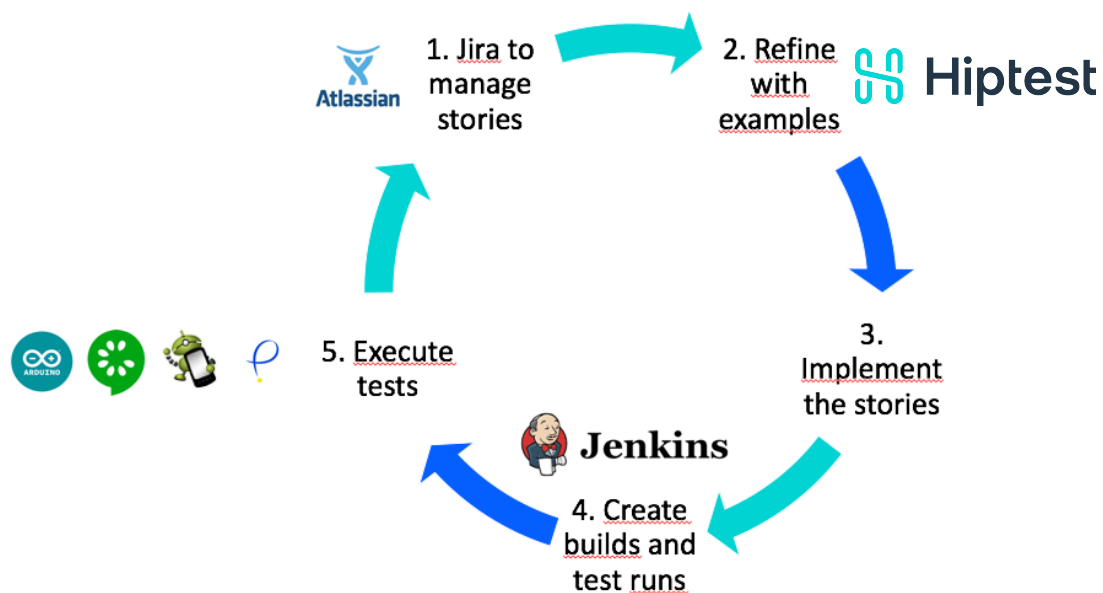
### a. Why BDD ?

Parkeon did choose BDD first because it enhances collaboration: they wanted to get people work together from six different countries. Moreover, they needed a common business terminology and BDD was the opportunity to create a specific wording that everyone could understand easily.

By using BDD, Christophe Rondeau, project director of Parkeon, wanted to create a shared understanding between the customer and teams (what are they doing, why and when).

### b. Process & tools

Concerning the process, Parkeon used the scrum with two weeks' iterations. The following tool stack was implemented:



### c. The test pyramid

At the functional level, the teams created 200 end-to-end tests, 940 system's integration tests and 1635 functional tests.

These 3000+ scenarios were all automated and just based on 1700 steps thanks to refactoring and step reuse! This was definitely key to automate the tests at scale.

### d. Outcomes

Parkeon's project was a success. They were mostly no delay in any of the releases even if 10% of the features have been changed. The project was completed on time which is amazing for such a long project.

The initial budget for testing was respected and there were no additional charges. Sometimes we see managers saying: "If we do test first, if we start with testing we won't be able to cut the test phase at the end of the project, so it will cost more." What is interesting here is that the amount of energy, of spending dedicated to the test activity, was exactly the same as other projects in Parkeon.



”

**BDD does not bring extra costs and dramatically streamlines the development process.**

**—Raphael Citeau, delivery manager at Parkeon**

# **A big thanks!**

**To our community and all the  
persons that shared their  
experiences and learnings with us.  
If you liked this reading share it!**