Behavior-Driven Development

What, why and how?
Table of contents

3 Introduction
4 What is behavior-driven development?
5 Traditional development vs. behavior-driven development
8 How does BDD support agile development?
9 BDD and no-code automation
12 Conclusion
Introduction

Every business today is undergoing digital transformation. Consumers are demanding high quality software solutions, putting pressure on businesses to deliver fast and frequent updates.

What does this mean for software development teams?

This means that traditional approaches to software development, where development and testing are siloed and don’t overlap, are outdated. Given that teams are expected to release quality software faster and more frequently, the practice of building software and testing it when the build is complete does not work.

In order to guarantee software quality at this accelerated development pace, the traditional software development process has shifted towards more agile methodologies. One of these is Behavior-Driven Development (BDD).

Even though this is not a new concept in software development, many teams still don’t leverage the framework to its fullest.

In this paper, we will give you an overview of what BDD is, why it is paramount for agile development and how no-code automation tools complement the methodology.
What is behavior-driven development?

Behavior-Driven Development (BDD) is a methodology in agile software development that aims to narrow the communication gaps between team members by encouraging collaboration between developers, QA and other members of the software development life cycle. This is done by using a commonly understood language and real-world examples to foster a shared understanding of how the application should behave.

BDD has emerged from Test-Driven Development (TDD), where tests are written before code, and code is written specifically to pass those tests. BDD makes use of the general techniques and principles of TDD while providing all relevant teams with a shared process to collaborate on software development.

This shared process is facilitated through the use of a simple Domain-Specific Language (DSL) using normal language constructs that express the behavior of the end-user and expected outcomes. By discussing end-user examples in this way, team members ensure a shared understanding on what kind of software to build.
BDD was developed as an alternative to the traditional waterfall method in order to better support agile development. So, what is the difference between traditional development and BDD?

Traditionally, in the software development industry, you would find teams with limited shared understanding of what was being built.

The traditional development process would go as follows:

1. **Relevant stakeholders** tell the product owners what their business needs are.
2. **Product owners** write requirements based on those business needs.
3. **Developers and testers** (independently) translate those requirements into code and test cases.
Misalignment would occur due to siloed interests:

- **Business stakeholder**: What problem do we want to solve?
- **Developer**: How do we build a solution that solves this problem?
- **Test manager**: What could possibly happen that prevents this problem?

BDD helps solve this problem by fostering better collaboration between all team members - product owners, developers, testers and stakeholders - through a shared language. A BDD process would look more like this:

1. **Relevant stakeholders** tell product owners their business needs.
2. **Product owners** then create a collaborative space with developers and testers so that they can talk about what they're going to develop together.
3. **During this phase they collaborate** around requirements and define them in plain English.
4. **Developers** write code based on these scenarios and **testers** create automated test cases that report back against these scenarios.

**Behavior-driven development process**

- **Business owner** defines business needs
- **Product owner, developer and tester** collaborate and define requirements together
- **Developer** translates requirements into code
- **Tester** translates requirements into test cases
- **Finished product**
This methodology allows the team to focus on the behavior of a system instead of the implementation features, thus driving value in the final product. By specifying these terms in advance, developers and testers won’t have to revisit work because requirements were misunderstood.

A way of formulating the end-user’s behavior is to create a shared language for your team that everyone understands. A scenario would be written using the following format:

- **Given** [initial context]
- **When** [event occurs]
- **Then** [ensure outcome]
- **And/But** [more outcomes if applicable]

By writing steps in this clear and simple way, everyone involved in the software development process can easily understand what’s going on and is able to contribute to the creation of behavior scenarios. This means that everyone in the organization is able to contribute their domain expertise to the project. Thanks to this collaborative environment, development becomes more efficient and error free – in other words, more agile.
How does BDD support agile development?

All in all, BDD offers a methodology that allows teams to become fully agile in the software development process. It extends and builds on standard agile practices, such as sprint planning, user stories and acceptance criteria, and makes them much more effective.

It helps teams improve product quality by fostering collaboration and enabling them to locate and concentrate on the features that really matter. As a result, siloes are broken down between teams and flexibility in development is enabled.

In sum, the following characteristics of BDD complement agile efforts:

- **Collaboration through face-to-face conversations**: BDD promotes communication between key stakeholders in software development.

- **Working software is the primary measure of success**: BDD encourages stakeholders to identify any potential misunderstandings and confusion early in the process, helping the team build the software.

- **Capture requirements at a high level – lightweight and visual**: BDD describes the end-user’s behavior through test-scenarios written in plain English.
In BBD, at the very beginning of a feature file, we write acceptance tests using the standard agile framework of a user story:

• As a user
• I want login to the shopping website with my username and password
• So that I will be able to enter the customer portal

Once the feature file has been established and agreed by the business participants, the acceptance criteria can be written to test that behavior. The acceptance criteria for a feature is written in terms of scenarios and implemented as classes. This allows for regular expressions – DSL – to map steps in a feature file to concrete actions. A scenario is written in the following format:

• Given Tom is on the login page
• When Tom enters his login credentials
• Then Tom should be able to log in

However, in order to deploy the test scenarios through an automation tool, we would in most cases have to translate them into code – meaning that implementation and maintenance would be in the hands of a few technical peers.

This presents a paradox: the bottleneck that we encountered when we first deployed agile development – and which we aimed to fix through BDD – is now present again due to coding requirements.

To solve this problem, no-code automation can be used.
The behavior format of BDD allows you to specify each statement in a modular way. Some tools, such as Leapwork, leverage a visual language rather than code that allows you to build test scenarios through modules that represent each behavior statement.

In the Leapwork Automation Platform, creating automated tests using a BDD methodology would look like this:

1. We write the BDD test scenario
2. We log into Leapwork’s automation platform
3. We create a step-by-step flow using visual flowcharts
4. We record the flow to see if it works as intended
5. Finally, we organize the flow by grouping sections into subflows that easily describe each of the actions of our test scenario

Following the previous test scenario example (Tom wanting to log in), the final flow in Leapwork would look like this:

![Flowchart](image)

Because each action – GIVEN, WHEN, THEN – needs more than one step to be completed, we have grouped those steps into subflows (shown in purple) to provide a simpler overview. The subflows can also be reused as is in other flows without recreating them every time they are needed.
Using a visual and codeless test automation tool such as Leapwork means that you can add one extra layer of visibility and understanding to the BDD process.

Test cases can now be used both as feedback on the application’s functionality as well as documentation on what the application does. BDD enforces team collaboration and cross-functional workflows, so it is very important that the test automation tool also facilitates those.
Conclusion

BDD is a method that fosters collaboration and enables agile development practices in software development. It encourages critical test analysis and design because developers and testers need to fully understand the desired end-product and how to test it before developing the code.

To further enhance BDD efforts, a no-code test automation tool can be adopted to add an extra layer of visibility and scalability to the QA process.

**Key takeaways:**

- BDD is a method that fosters cross-functional team collaboration
- BDD makes use of domain-specific language (DSL) that can be easily read and understood by everyone
- Test cases clearly articulate the requirements for the code
- No-code test automation tools complement BDD by using an intuitive visual language for test case building. This means that test cases are easily understood by all in the software development process.
Learn more about no-code test automation for agile teams.

Book demo  Start trial