WHAT IS BDD (BEHAVIOR-DRIVEN DEVELOPMENT)?



Widely known in the tech world by its abbreviation BDD or by the nickname “Specification by Example”, Behavior-Driven Development is generally defined as “a methodology for developing software through continuous example-based communication between developers, QAs, and BAs.” Created by a man named Dan North, BDD is his answer to the frustration of not being able to explain clearly when developers wanted to know where to start, what not to test, how much to test, what to call the tests, and how to understand why a test fails. Stemming from Test Driven Development (TDD) and Acceptance Test-Driven Development (ATDD), this methodology is used to bring agility, concrete communication, and a shared understanding of objectives between teams, technical and non-technical.

Essentially the collaborative process of describing, in a very direct example-driven way, a set of behaviors that can be expected from a system–ultimately, the outcome is that all stakeholders are able to have powerful conversations about what the software should do. On top of that, when implemented correctly, everyone in all teams will understand the functionality of deliverables without confusion and be able to identify key results before the development process even begins. In the following article, we will explore the ins and outs of BDD as well as examples of how to implement it into your organization’s software development management methodology.

BDD in Agile

BDD’s collaborative software development approach works well with the short sprints, continuous feedback, and iterative approach to improvements of Agile methodologies. One doesn’t replace the

other, but instead they align to enhance one another. BDD provides Agile processes with a structured, human-readable way to define requirements and describe scenarios and user interactions so that developers have a customer-centric, business-focused, real-world understanding of what the software needs to do.

One way to think of BDD is as a collection of plugins for improving your Agile processes. When BDD and Agile integrate, your organization can quickly respond as needs change to truly deliver on the Agile promise of reliably working software delivered when needed, while supporting maintenance and keeping teams focused and disciplined.

BDD Process

BDD is a process involving three steps that flow continuously and repeatedly to make bite-sized changes that are more manageable. The idea is to move through these steps quickly, to document the wins so that they are repeatable, and to strengthen your process with each iteration.

- **Discovery:** The process begins with developing the "user story" that describes what the user needs to do, the functionality they require, priorities, and possible gaps in understanding. The goal is a mutual, complete, and detailed idea of needs and the reasons behind them. In BDD, these stories have a structure, as you will see in the examples.
- **Formulation:** The next step is to document the takeaways from the user stories, essentially feeding back to the users and other key stakeholders what the developers understand. Formulations are executable specifications that can be shared for feedback and revision. The result is a shared, mutually understood vision of what to build.
- **Automation:** This is the stage where the software specifications are implemented into code as executable test scenarios that describe how the system should work, in human-readable language. These scenarios or automated examples make it easy to understand what the software does.

The process emphasizes making small changes, iterating rapidly, and continuously adding value to the system while being ready to respond to feedback.

Acceleration of TDD and ATDD through BDD

In order to understand BDD, you must first be introduced to the meanings of TDD and ATDD as well as how they are used. Following that, you must then understand how BDD accelerates these two methodologies by combining their basic principles into one powerful software development process.

TDD "refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests), and design (in the form of refactoring)." When this methodology is in use, a developer must first define a test set for a specific unit, then purposely make it fail, then deploy the unit, followed by defining if the unit was a success. A very key part of the success of BDD, a programmer should know and work within the TDD structure before learning or implementing BDD.

ATDD "involves team members with different perspectives (customer, development, testing) collaborating to write acceptance tests in advance of implementing the corresponding functionality. The collaborative discussions that occur to generate the acceptance test are often referred to as the

three amigos – representing the three perspectives of customers (what problem are we trying to solve?), development (how might we solve this problem), and testing (what about…).” Each test within this methodology should be written in a “given, when, then” style or if-then format, like below:

Given

When

Then

With the first focus on reducing overall development time as well as the number of bugs found via tests on all requirement levels and stronger codebases, the second focus zeroes in on simplifying communication via a Domain-Specific Language format and the interworking of every end result perspective, when combined we get BDD. Now, taking what we know from above–that BDD is an example-driven communication methodology used to talk about application behaviors–at its core, BDD basic principles use the conversation structure that even non-developers can understand from ATDD in conjunction with a narrowed down TDD testing focus aimed at behavior specifics that are best described in business-driven logic.

The 2 Basic Principles

1. User Stories

Plain language written scenarios that describe the intent, the who, and the what should happen in order to achieve the requirements of the unit. Within BDD, User Stories need to be very specific and always in whole sentences including the name/feature. When introducing the method, Dan North stated that “Developers discovered it could do at least some of their documentation for them, so they started to write test methods that were real sentences. What’s more, they found that when they wrote the method name in the language of the business domain, the generated documents made sense to business users, analysts, and testers.” Ideally using the word “should” and in an if-then format, BDD technically calls for no specific scenario format. However, many experts, including Dan, suggest that your organization work with one standardized format. That way teams can modify, explore, and expand without any issue, now and in the future.
Dan North’s User Story Format

Title (one line describing the story with should)

Narrative:
As a
I want
So that

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given
And …
When
Then
And …

Scenario 2: ...
To help you better understand how it looks, with slight variations of format, here is an API test example taken from Guru:

Feature: Test CRUD methods in Sample REST API testing framework

Background:
Given I set sample REST API URL

Scenario: POST post example
Given I Set POST posts API endpoint
When I Set HEADER param request content type as "application/JSON."
And Set request Body
And Send a POST HTTP request
Then I receive valid HTTP response code 201
And Response BODY "POST" is non-empty.

Scenario: GET posts example
Given I Set GET posts API endpoint "1"
When I Set HEADER param request content type as "application/JSON."
And Send GET HTTP request
Then I receive valid HTTP response code 200 for "GET."
And Response BODY "GET" is non-empty

Scenario: UPDATE posts example
Given I Set PUT posts API endpoint for "1"
When I Set Update Request Body
And Send PUT HTTP request
Then I receive valid HTTP response code 200 for "PUT."
And Response BODY "PUT" is non-empty

Scenario: DELETE posts example
Given I Set DELETE posts API endpoint for "1"
When I Send DELETE HTTP request
Then I receive valid HTTP response code 200 for "DELETE."

2. Common Languages

As we have stated numerous times throughout this article, and can not stress enough, BDD relies heavily on the use of language that every member of your team can understand. Domain-Specific Language (DSL) is what makes each scenario readable by business partners, the ones who are commissioning the software and tests in the first place, wanting a specific behavior to occur. It works to remove the need to explain, on both technical and non-tech ends, how a behavior is implemented and in return makes less confusion throughout the process.

Tools

In order to maximize time and outcomes, BDD is supported by tools that automate tests. Closely linked to the DSL that is defined by your organization, these tools include Cucumber, which can understand Gherkin while supporting writing specifications for 30 spoken languages. When in use,

such tools automatically execute common language specifications.

Launch Behavior-Driven Development In Your Organization

No matter where your organization is in its software development management journey–using agile methods, waterfall, or something else–implementing this methodology yields a higher gain than risk value. If the use of the method does not suit your needs, you can simply go back to writing test requirements the way you have been doing it. There is no undoing anything.

To begin, first, ensure your developers are working within and understand TDD. If they are not, they will need to start a training period to develop skills within this style of programming. Once your developers are on board, you can implement the basics of BDD to your entire organization. Again, a training period will need to take place. During this time Developers, QA, and BA teams need to be taught how to read and understand your chosen DSL and documentation method. Start with just one project to see how it all works. Be aware that this methodology requires specifications before development and constant outside feedback from users, customers, and domain experts. However, when implemented successfully, you will reduce regression and improve goal communication.

Benefits of BDD software

Widespread adoption of BDD for software development is being driven by its effectiveness in translating business needs into executable software code. BDD provides:

- **Enhanced communication:** The approach makes it possible for all stakeholders to describe their needs and expectations in normal language, for clarity that is impossible with technology jargon or programming code.
- **Shared understanding:** Because all stakeholders, including non-technical users and decision-makers, can understand all phases of discovery, formulation, and automation, projects start and move forward with a clear and common understanding of what the software will do.
- **Early detection of issues:** When you "shift left" to test, find, and fix issues early through example-driven communication, you streamline and cut down on time, frustration, and costs.
- **Improved test coverage:** Because test writing is so much easier, you can address all possible scenarios for more thorough and relevant testing.
- **Reduced rework:** With better clarity around and understanding of requirements, fewer mistakes happen.
- **Faster feedback loops:** Writing and automating scenarios, with solid testing and feedback, is easier and faster with each iteration.
- **Alignment with business goals:** Because non-technical stakeholders are integrated from to beginning, the development process stays aligned with business objectives.
- **Ease of maintenance:** Scenarios don't just guide development, they become self-documenting test cases so it is easier to maintain and update software functions.

Common pitfalls of BDD approach

No approach to software development is perfect, and that is true for BDD. You may encounter some snags and complications when adopting and using it. These include:

- **Steep learning curve:** You and your teams will need to be knowledgeable about more

concepts than needed for TDD. Novices will benefit from having experience with TDD before taking on BDD.

- **Tool dependency:** There can be a tendency to focus on tools without enough attention to the principles, the user story of BDD. Keep the focus on collaboration, communication, mutual understanding, and better test coverage.
- **Conceptual misalignment:** Teams may not understand the underlying purpose of BDD, thinking of it as being about technology versus seeing it as a collaborative approach. The result can be an overreliance on tools and testing frameworks, missing that it is, at the heart, a way to improve communication across tech and non-tech stakeholders.