Add behavior-driven development to your Java Agile development toolbox

Behavior-driven development is the next natural step from test-driven development.

by Sashank Chithajallu

August 27, 2021

Agile is often considered the holy grail of product-development methodology, since it provides greater flexibility to adapt to changing business scenarios and requirements. In this article, I discuss behavior-driven development (BDD) as the next step beyond the popular test-driven development (TDD) Agile methodology.

Many product-development projects use Agile, which ensures that the product optimally meets the requirements of the customer and user. Unlike the waterfall model of software development where all the requirements are known, detailed to a granular level, and frozen beforehand, Agile allows the requirements to be fluid and therefore evolve during product development.

This is not to say that Agile is always superior to the waterfall model—each has its own purpose and application. A waterfall model may be well-suited for the development of complex mission-critical systems, where each module or submodule is developed in isolation across geographies or companies. Having a predefined and fixed requirement set ensures that the integration between these modules is seamless. But, in any methodology, the product-development cycle always starts with requirements.

It starts with the customer or end user

Generally, the requirements are dictated by what the customer or end user of the product is expecting, usually in the form of a business ask conveyed in simple vernacular.

The product manager interfaces between the users and the development team and translates these business asks into business
The conformance of the product’s functionality to a customer ask is established with testing. It is important to understand that testing is not just a methodology of certifying a product’s conformance to requirements but part of a holistic product-development process. The goal of testing is to reduce the amount of time spent on reworking and fixing issues while ensuring that the established requirements are met.

As Agile methodology allows for constantly changing requirements, the standard design → develop → test cycle may no longer be sustainable and may result in identifying issues late in the development → test cycle, causing major rewrites and late deliveries. Agile is all about fast-paced development that allows working features to be delivered to a customer in iterations.

That’s where behavior-driven development comes in.

Before we understand BDD, we need to understand its roots in test-driven development, which saw a re-emergence during the Agile software development movement.

**First: Test-driven development**

American software engineer Kent Beck is often credited with the rediscovery of this methodology. TDD has its origins in the test-first programming concepts of extreme programming, which was developed by Beck while working on the Chrysler Comprehensive Compensation System payroll project. (See *Java Magazine’s* [interview with Kent Beck.](#))

Some experts maintain that the methodology dates to the mainframe era, when developers would come in with the expected results of a program and develop → run each program and then compare the output with the expected result. This would lead them to correct the program if any discrepancy was found with the expected results. In a way, the test is created before the program code itself.

In TDD, the test cases are developed from the requirements, even before the actual program is created. Then the program is iteratively developed based on the requirements while simultaneously being validated against the tests that were created. The program is fixed or enhanced continuously with testing, with the goal of ensuring that all the created tests are executed successfully.

The following is considered a generic cycle for TDD of a given feature or requirement:

1. Develop a test. Based on use cases and user stories, a test is written such that it passes if and only if the feature’s specifications are met. In other words, if the feature being developed conforms to the requirements, the test should pass.
2. Run the tests and observe any failures. The test run should fail if and only if the feature doesn’t meet the requirement.
3. If any tests fail, develop or correct the code in the simplest sort of way to allow all the tests to pass.
4. Iterate steps 2 and 3 until all the tests pass.
5. Refactor the code to maintain functionality while improving compliance with standards for abstraction, modularity, and readability. During the process, continue running the tests to ensure that functionality has been maintained.

The five-step process is carried out for each of the features that are a result of one or more requirements. The test development may itself be carried out in an incremental manner alongside the code development, so long as the test is written before the program code.

The important thing to note is that in TDD, the feature to be tested needs to be small. If the feature is big, it needs to be broken down into pieces to ensure that each test is small and easy to understand while reducing the development or debug times.

The next step: Behavior-driven development

BDD should be considered as the next logical step and an extension of TDD. Dan North pioneered BDD through his article *Introducing BDD*.

BDD follows that same principle of creating tests before development and validating the code against the tests repeatedly until every test passes. However, TDD doesn’t specify what exactly a test may be defined on, whereas in BDD, each test is required to be defined in terms of a desired behavior of a feature.

BDD refines TDD by incorporating concepts from domain and object-oriented design, in which the design speaks the language of the business domain. Specifically, the code structure and modules align neatly with the business modeling of various entities. This provides for better communication and collaboration between all the stakeholders involved. Customer and end-user requirements can be easily translated into the technical design.

BDD is based on this principle of testing features based on what the business requirement specification desires. In Agile, the desire of the requirement or customer’s ask is defined as a user story, which can be considered a behavior specification for a feature. BDD draws out tests directly from these Agile user stories that need to be created through collaboration between the product manager, the customer or end user, and the developers.

User stories in BDD

A user story is an expression of the functionality that a product feature needs to provide from the perspective of a user. The purpose of a user story is to define how that feature can or needs to provide value to the user. Thus, the user story ensures that the end user’s business requirements are brought to the focus for developing and delivering a feature.

It follows that any feature developed to cater to that user story needs to be tested against the specifications of the user story. And this is exactly what BDD does. It brings the user’s perspective into the testing of a feature, which ensures that the feature meets all the desired behavior patterns of the requirement. By using the constructs of business language, BDD reduces or eliminates the “lost-in-translation” effect sometimes found in requirements-to-tests conversion.

A simple user story may be defined as follows:
Title
The title of the story

Description
A short description of what the story is about in the below format

- **As a:** “User who is looking for or defining the value of the feature”
- **I want:** “The desired behavior of the feature that can add value”
- **so that:** “The value provided by the feature”

Acceptance criteria
A description of various behaviors that make the story acceptable for usage. There may be multiple acceptance criteria for a given user story that are defined as multiple scenarios in the below format:

**Scenario 1:** “Brief description of the expected behavior”

- **Given:** The current/specified context
- **When:** Some action is performed
- **Then:** The expected outcomes that can be observed

The language used in defining a user story and the corresponding acceptance criteria is semiformal, which ensures that both technical and nontechnical personnel can define, read, and understand it.

Acceptance criteria is a core construct of the business requirement that defines the behavior that the user expects from the feature. BDD heavily depends on the acceptance criteria defined in the user stories; acceptance criteria defined in the Agile user story drives the tests that validate the feature developed to cater to the user story’s requirement. These acceptance tests determine whether the feature that was developed meets the specified requirements.

The obvious question now is, “How do we convert the acceptance criteria into measurable or executable tests?”

**BDD and test automation**
As with TDD, automated tests are core to the BDD methodology. But, unlike TDD, where the acceptance criteria are not required to be defined in a specified format, the acceptance criteria in BDD are defined in the *Given [] When [] Then []* format. And this is where the specialized tooling and test automation frameworks come into the picture.

In BDD, since the specification format for acceptance criteria is clearly defined, the tools are usually designed to leverage the semiformal language of the acceptance criteria specified in the user stories. The specialized tools convert the requirements defined in the *Given [] When [] Then []* format into executable tests.

**JBehave** is one such tool developed by Dan North. These specialized tools require the project team to define snippets that are used to transform the steps in the acceptance criteria into executable tests.
Example: In the case of JBehave, if you refer to the Getting-Started page, you can find that the story is written into a file that has the .story extension. The Story is written in semiformal text in the Given, When, Then format. Once the story is created, there needs to be a plain old Java object (POJO) that holds the Java methods that can be mapped to the steps in the story.

The Story by itself is not executable and needs to be transformed into executable code to be run. The POJO contains the necessary executable code that can be mapped to the steps using supported annotations. In the case of JBehave, the annotations @Given, @When, @Then, and @Aliases help define the mapping between the textual story and runnable code.

JBehave isn't the only tool for BDD. Another is Cucumber. Though Cucumber was originally written for Ruby-based application testing, it now supports additional languages including Java and JavaScript. Cucumber uses Gherkin language to define test cases, which is designed to be human-readable.

The overall approach to BDD using Cucumber and Gherkin is the same as mentioned with JBehave. Cucumber tests are classified into features, and each feature has one or more scenarios with a specified set of steps in each. The scenario definition follows the Given{}, When{}, and Then{} approach. These steps are linked to a step definition written in a programming language.

**Natural language processing**

If you are feeling adventurous and want to try and create your own smart custom testing tool or implementation for BDD, you could try using machine learning. Natural language processing (NLP) may be a good place to look, as it can help enhance your BDD test definitions over time.

NLP also allows you to perform document categorization. Though this may not be the true purpose of document categorization, a simple implementation could be done using Apache OpenNLP Document Categorizer.

You can define a training file containing various phrases that may be used in steps in a test definition using the Given, When, Then format. These phrases can be mapped to a code-based definition or method name of what the phrase is expected to do. So, when a user defines the acceptance criteria using certain phrases, the corresponding tests can be generated or set up at runtime, based on the phrase → method mapping, and the NLP engine can determine which method to invoke based on the model generated by the training. The model can be retrained on new data sets, which can be gathered from regular usage.

The NLP engine could be trained with a base set of training data that would contain a phrase-to-method mapping. Once the NLP model is trained, it can process informal acceptance criteria and should identify the methods, given the phrase in the acceptance criteria. The NLP could prompt or provide a method, given a phrase, but if the user disagrees with the method used, then the training file can be updated, and the model can be retrained using the latest information. A BDD engine based on such a premise could evolve over time.
For example, consider a single step where the program needs to ascertain if a user is logged in to the application. Assume there’s an existing `ConfirmUserLoggedIn` method to verify that the user has logged in.

The step may be specified in the informal syntax as “Given a logged in user,” or “Given a user is already logged in,” or “Given a user is authenticated,” and so forth.

In the above context, both *logged in* and *authenticated* mean the same thing. So, if you have trained the model by providing the mapping between the *logged in* phrase and the `ConfirmUserLoggedIn` method, and if the BDD’s acceptance criteria contains “Given a user is authenticated,” then the NLP may not provide the expected results. This can then be corrected by adding this new phrase into the training file by linking it to the `ConfirmUserLoggedIn` method, and then retraining the model. The next time the NLP-based implementation sees this new phrase, it can map it to the correct method.

Also, when NLP is used, the phrases need not be an exact match, which adds greater flexibility to the user when writing acceptance criteria. There is obviously a lot to this approach in terms of what can be done and how, but I will leave it to your imagination as to how this can be implemented and extended.

### A few key points about BDD

Keep the following in mind:

**BDD is not just about tools and testing.** As with Agile, it’s a way of doing things and involves people and processes—simply put, it’s a way of life. Starting from the customers and end users who have a need, to product managers who define the requirements, to the developers, to the testing personnel who ensure the requirement is met, it’s necessary that they understand and believe in BDD.

**The entire development process is interdependent.** The requirements are formulated into informal acceptance criteria that in turn lead to the feature development and testing. The BDD process requires participation from all phases and people in the software development lifecycle.

**It’s all about collaboration.** The BDD-based approach requires that users, product managers, developers, and testers together formulate the specifications and ensure that they are aligned with respect to any given feature. It would be wise to have these meetings toward the start of the project, as any hiccups can be identified up front, which is the whole point of BDD.

**Consider the test automation tools’ requirements.** As with any product development, it is very crucial to develop an application’s architecture taking into consideration the test automation’s architectural requirements, such as web services-based test automation versus GUI-based test automation. This will help ensure that the test automation suite can be developed for testing the application and will accelerate the BDD approach.

**Consider microservices.** Because many applications being developed today use microservices-based architecture, automated testing through the web services channel is gaining a lot of popularity. Testing an
application using web services is faster and more reliable when compared to GUI-based test automation, as an application’s GUI tends to change in form and structure more often than the underlying service.

**Don’t abandon the GUI tests.** GUI-based testing also adds value because proper user interaction through the GUI is one of the prime considerations for an application. It is imperative to come up with a consolidated testing strategy and plan to accommodate for both approaches, with a probable focus on web services-based test automation. This can then be incorporated into the way acceptance criteria is defined so the BDD can be used.

**Define your language.** It’s important to come up with the terms that may be used in the informal definition of the acceptance criteria. This is required specifically so that all parties involved understand and speak the same language, which will further the acceptance criteria to the automated test conversion process seamlessly.

**Start from the beginning.** BDD can help reap more benefits if it is implemented in a project right from the start and with the complete agreement of all the personnel involved. Trying to align an existing project with BDD may cause a lot of work—and you may find yourself in the middle of an unhappy team.

**Conclusion**

The BDD approach is not a complete replacement for your project management methodology or documentation, but you should think of this as an augmentative process that would help identify more issues well within an Agile development lifecycle, thereby reducing the risk later in the project’s lifetime.

The biggest advantage with BDD is the traceability it provides between the requirements and tests as well as the reduction in loss of information in transmission between a requirement and its corresponding test.

While moving to a BDD-based approach is considered advantageous, it is also necessary to consider the resource and skill availability in addition to the overall processes currently being followed in your organization.

Adapting to any change takes time, and BDD is about constantly adapting to constant change. But with the right mindset and a cohesive set of people, it can spell the winning formula for your project.

**Dig deeper**

- *Java Magazine* interview with Kent Beck
- *A Philosophy of Software Design*, reviewed by Andrew Binstock
- Refactoring Java, Part 1: Driving agile development with test-driven development

**Sashank Chithajallu**

Sashank Chithajallu is a principal product manager at Oracle, where he leads the technical strategy on the company’s offerings for energy
and water providers. His current area of work is in SaaS technologies, including Kubernetes. Chithajallu works in Java, Spring Boot, Cordova, Python, and SQL.