# Behavior Driven Development in Modern Complex Systems

Ibrahim Abdelkareem Ibrahim Abdelkareem

May 23, 2023

## 1 Introduction

Behavior-Driven Development [1], introduced in 2006 by Dan North and inspired by Test-Driven Development [2] is an agile software development process that aims to reduce the communication gap between business analysts, testers, and software developers by defining the criteria to be tested in order to consider the feature implementation successful (i.e., acceptance tests) in a business-readable and domain-specific language (DSL) [3] called Gherkin to describe a system behavior while hiding its implementation details. In BDD, scenarios for acceptance tests are expressed with the following simple syntax:

```
Given some initial context
When some event happens
Then ensure some outcome
```

BDD encourages using scenarios in the above format to gather software requirements. The keywords Given, When, and Then in the code snippet above are known as "Step" in BDD. Each step is parsed and executed by a BDD framework (e.g., Cucumber) to verify that the software features meet business expectations, which is known as Automated Acceptance Tests (AAT). Scenarios written with the Gherkin syntax empower communication and collaboration among developers, testers, and business analysts to improve the overall quality of the software. They also act as living documentation and references for the expected behaviors of the system. There are many tools to help developers convert their scenario specifications into a documentation website, such as Cucumber LivingDoc.

In this essay, I'll demonstrate a practical introduction to behavior-driven development in which I explain how business analysts collect requirements and create scenarios using Gherkin language syntax and how developers make these steps executable using step definitions. I'll also briefly show how SpecFlow, a BDD framework, works. Then I'll discuss two papers regarding the application of behavior-driven development in complex systems such as financial trading and automated driving and explain how behavior-driven development helped them to lift the communication barrier between professionals from different disciplines, to overcome the difficulties with automated testing and documenting the features of their systems, and how the reusability of BDD opened the possibility to cover more test scenarios while reducing the overhead of maintaining the automated tests.

## 2 Practical Introduction to BDD

### 2.1 Defining Scenarios

The industry's adoption of agile methodologies such as TDD, BDD, and Acceptance Test-Driven Development (ATDD) is growing due to their promise of helping projects succeed. The interest that has been growing for BDD specifically within the industry is due to its capability of engaging business analysts to gather requirements and convert those requirements into executable scenarios that work as automated acceptance tests while still being readable for business people. In BDD, business analysts would collect requirements and write them in gherkin syntax as follows:

```
As a KTH student
I want the ability to login to the student web
So that I can browse the digital library

Scenario: User login with valid credentials
  Given I'm a KTH student
  And I have a valid credentials
  When I login to the student web
  Then I should be able to browse the digital library

Scenario: User invalid credentials shouldn't be allowed to login
  Given I'm a KTH student
  But I have an invalid credentials
  When I login to the student web
  Then I should see invalid credentials message
```

The first section in the code snippet above is the User Story, a short and simple description of the features of the software being tested. A User Story is validated by multiple scenarios, each of which covers and validates a certain use-case (e.g., using valid credentials, using invalid credentials, etc.). In BDD, this code snippet is typically stored in a file called Feature File (e.g., login.feature) that lives in the source control next to the software code and is executed as part of the CI/CD pipeline to ensure that every revision of the software is validated against the business requirements.

```
}
```

*Note the extra two keywords And and But in the snippet above. These are supporting keywords that can come after a Given, When, or Then to help analysts add more context for each step in a different line in order to support readability.*

## 2.2 Creating Step Definitions

After the requirements have been collected by the business analyst and discussed and refined with developers, they will generate a Step Definition for every step in the feature file. A Step Definition is an executable code that's responsible for executing the logic behind every step in a scenario. There are many frameworks that integrate with many programming languages to help developers write Step Definition in their favorite programming language and link them to the feature files. Examples of such frameworks are Cucumber, SpecFlow, and JBehave. The Step Definitions for the code snippet above in SpecFlow will look like.

```
[Binding]
public sealed class KthLoginStepDefinitions
{
    private readonly ScenarioContext _scenarioContext;

    public KthLoginStepDefinitions(SenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }

    [Given("I'm a KTH (student|teacher)")]
    public void IAmKthStudent()
    {
        // Implementation goes here
    }

    [Given("I have a valid crednetials")]
    public void IHaveAValidCredentials()
    {
        // Implementation goes here
    }

    [Given("I have an invalid credentials")]
    public void IHaveAnInvalidCredentials()
    {
        // Implementation goes here
    }

    [When("I login to the student web")]
    public void ILoginToTheStudentWeb()
    {
        // Implementation goes here
    }

    [Then("I should be able to browse the digital library")]
    public void IShouldBeAbleToBrowseTheDigitalLibrary()
    {
        // Implementation goes here
    }

    [Then("I should see invalid credentials message")]
    public void IShouldSeeInvalidCredentialsMessage()
    {
        // Implementation goes here
    }
}
```

Step Definitions are linked to Steps in the feature file by annotating functions with a regex pattern, as shown in the code above. When a scenario test runs, the BDD framework will call each Step Definition for the scenario step in the specified order. Step Definitions are stateless functional units, but they can still share data via a common container, such as ScenarioContext in this case.

## 3 Re-usability in BDD

While there are many advantages to using BDD, such as documenting the functionality of a system in a series of executable acceptance tests and increasing the collaboration between professionals from multiple disciplines, one of the major advantages that mainly benefits complex, distributed systems is reusability.

As Gherkin syntax describes scenarios with a series of steps that can be reused by different scenarios in the same test suite *(e.g., I'm a KTH student in the previous section)*, developers can create packages or steps repositories [4] to contain system-wide shared steps (e.g., authentication, defining actors, ...etc.).

This re-usability feature is very powerful and, when put in use correctly, can drastically improve the testability and confidence of very complex systems, as we'll explore in the next section.

## 4 BDD Application in Modern Complex Systems

### 4.1 Testing Complex Microservice Architecture via BDD

In 2021, the Danish investment bank Saxo Bank published a paper in the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW) [5] to demonstrate how BDD helped them test and document the functions in their complex system.

Saxo Bank has a highly complex API-first architecture backed by an increasing number of micro-services that expose functionality to trading in the financial market via REST or gRPC endpoints. In the first version of their APIs, they relied heavily on unit tests written solely by developers (e.g., unit tests) and manual tests executed by testers, which limited their ability to have a higher quality system with a fast release cycle as their automated tests suffered from nondeterministic results and didn't convey what was tested and what wasn't tested from a functional point of view, and such knowledge tend to get lost in the complexity of systems when isn't well documented over time. With that said, even a successful test run won't provide developer and stakeholders with high confidence in their system.

In the paper, the authors focused on the idea of capturing the initial context. The initial context is a concept in BDD that usually defines the actor(s) of the scenario with the state and permissions that are required by the test case. The initial context is typically captured by the given keyword *e.g., Given a KTH student with valid credentials, as seen in the example in section 2*. The authors focus on the initial context was because they found that most of their problems were a result to the lack of explicit focus on initial context (IC). They found that, when a test is flaky it is often because the initial context state in the test environment isn't what the test case expects. They also found that the functional requirements aren't clear because they didn't perform systematic analysis involving the initial context to describe which actors, uses which features and how they should use it. So they came up with an approach for test analysis in which they include the initial context in the input space when analyzing the functional coverage for the APIs, so that it influences the outcome of the test case.

While using the initial context seems to solve a lot of issues, establishing the initial context is generally a hard problem. In the paper, the authors suggests focusing on the bounded-context *a concept from domain-driven design* [6] is a practical way to establish relevant initial context for the test case.

The conclusion of this paper is that using BDD not only improved collaboration between people from different disciplines but also helped to regain confidence in the system and discovering its features as it provides a method to collect acceptance test requirements in a natural language that can be executed and verified automatically via the CI/CD. The main takeaway from their suggested method is that defining the right initial contexts for different functions and injecting them in the input space for automated tests eliminates the risk of flaky tests and also helps to document the functions with respect to their right initial context (e.g., actors).

## 4.2 Scenario-based Verification of Automated Driving via BDD

In 2022, a paper was published in IEEE 25th International Conference on Intelligent Transportation Systems (ITSC) [7] to emphasis on the benefits of incorporating scenario-based verification using BDD and the gherkin syntax in the automated driving industry.

The authors discuss the difficulty of going to production with automated driving software due to the complexity of the technology and the involved organizations. In the industry, there are two major approaches to managing this complexity. System engineering methodologies are applied to formalize development processes and artifacts, and agile methodologies are applied to manage interdisciplinary teams. One of the major challenges both approaches are trying to resolve is the verification and validation of automated driving software systems. Each of these approaches has different ways of tackling this challenge. System Engineering rely on divide-and-conquer thinking to specify the system boundaries within which the system is designed to operate (i.e., Operational design domain, ODD) as defined in ISO 214481 [8]. Agile development focuses on enhancing processes so that teams can introduce and ship small product increments to avoid shipping large solutions in one go to production.

There are many different ways to validate a system with regard to a specified ODD. The author believes that scenario-based systems engineering (SBSE) [9], an approach in virtual verification looks promising and is well-established. The main idea of SBSE is it uses logical scenarios from refining system boundaries and formalizing traffic scenarios which specify development and verification processes. Logical traffic scenarios have a multi-dimensional continuous parameter space (e.g., velocity of traffic participants, curvature of road geometry, ...etc.) which are described by a parameter range, so in theory, there is unlimited possibilities of different parameter combinations which makes testing and defining all scenarios hard. The author suggests having function-independent scenarios, as the expected behavior of a function in a scenario, doesn't necessarily match the outcome of every other function with respect to the same scenario parameters.

The author used BDD to describe the system's expected behaviors and requirements by relying on two main artifacts. The logical scenario specification is a function-independent formal description of the ODD from which concrete scenarios are sampled, and the gherkin test specification is a function-dependent formal description of the AD system's desired behavior. BDD frameworks will execute the Gherkin syntax step-by-step, which in this case translates the automated driving requirements into time-spatial interval descriptions and applies them to function-independent scenarios. Doing so opens the possibility of having an independent logical scenario database that can be built and updated by the different sub-system functions. It also allows for random scenario generation and the automation of acceptance testing of the complete requirements, rather than focusing on the critical metrics only. All scenarios can also be reused and executed against all test definitions, which may help find potential faults in the system.

Not only did BDD help the authors have human-readable tests with reusable components, but it also increased the collaboration between system engineers and test engineers from the early development stages, which yielded higher-quality software that met behavioral expectations to a greater degree.

## 5 Conclusion

BDD is a well-established agile methodology that focuses on describing the behaviors of the system in a series of executable acceptance tests. These acceptance tests can be executed in the CI/CD to validate that the software being tested adheres to all business rules and also serve as living documentation and a reference to the system's functions. BDD also engages people from different disciplines to define the behaviors of the system, which leads to fewer communication round-trips and reduces the communication gap and misunderstandings.

BDD's end goal isn't only to define a series of automated acceptance tests written in natural language; it's also a process to demystify the complex flows of complex systems to support the understanding of each function and its intended actors, which leads to a higher degree of confidence working with the software, as we saw in Saxo's bank paper.

BDD has great support for reusability, which easily allows developers to have shared packages or repositories for system-wide steps or can even randomize and generate steps for complex systems such as automated driving software, as we saw in Section 4.2.

## References

[1] *Introducing BDD*. en. Sept. 2006. URL: `https://dannorth.net/introducing-bdd/` (visited on 05/21/2023).

[2] Kent Beck. *Test-driven development: by example*. The Addison-Wesley signature series. Boston: Addison-Wesley, 2003. ISBN: 9780321146533.

[3] *Business Readable DSL*. URL: `https://martinfowler.com/bliki/BusinessReadableDSL.html` (visited on 05/21/2023).

[4] Mazedur Rahman and Jerry Gao. "A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development". In: *2015 IEEE Symposium on Service-Oriented System Engineering*. 2015, pp. 321–325. DOI: `10.1109/SOSE.2015.55`.

[5] Brian Elgaard Bennett. "A Practical Method for API Testing in the Context of Continuous Delivery and Behavior Driven Development". In: *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2021, pp. 44–47. DOI: `10.1109/ICSTW52544.2021.00020`.

[6] Eric Evans and Martin Fowler. *Domain-driven design tackling complexity in the heart of software*. Addison-Wesley, 2019.

[7] Christoph Lauer and Christoph Sippl. "Benefits of Behavior Driven Development in Scenario-based Verification of Automated Driving". In: *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*. 2022, pp. 105–110. DOI: `10.1109/ITSC55140.2022.9922498`.

[8] *ISO 21448 - Road vehicles — Safety of the intended functionality*. June 2022. URL: `https://www.iso.org/standard/77490.html`.

[9] Christoph Sippl et al. *Scenario-based systems engineering: An approach towards automated driving function development: Semantic scholar*. Jan. 1970. URL: `https://www.semanticscholar.org/paper/Scenario-Based-Systems-Engineering%3A-An-Approach-Sippl-Bock/f9b4924b6f72f40911990d590fa50e49b9278e6e`.