The Report Analysis and Characteristics of Behaviour Driven Development with Cucumber Model

[1]Dr. Ravi Saripalle, [2]Buddharaju Shanmukh Varma,
[3]Raghukanth Reddy Gudimetla, [4]Sudeepa Gorle
[1,2,3,4]GVP College of Engineering (A), Visakhapatnam, AP, India

Abstract
Behaviour Driven Development (BDD) has gained increasing attention as an agile development approach in recent years. However, characteristics that constitute the BDD approach are not clearly defined. In this paper, we present a set of main BDD characteristics identified through an analysis of relevant literature and current BDD toolkits. Our study can provide a basis for understanding BDD, as well as for extending the existing BDD toolkits or developing new ones.

Keywords
Behaviour Driven Development, Test Driven Development, Ubiquitous Language, Automated Acceptance Testing, cucumber.

I. Introduction

Behavior Driven Development (BDD) is an increasingly prevailing agile development approach in recent years, and has gained attentions of both research and practice. It was originally developed by Dan North [3] as a response to the issues in Test Driven Development (TDD).

TDD is an evolutionary approach that relies on very short development cycles and the agile practices of writing automated tests before writing functional code, refactoring and continuous integration [19]. Acceptance Test Driven Development (ATDD) [1-2] is one type of TDD where the development process is driven by acceptance tests that are used to represent stakeholders' requirements. ATDD helps developers to transform requirements into test cases and allows verifying the functionality of a system. A requirement is satisfied if all its associated tests or acceptance criteria are satisfied. In ATDD acceptance tests can be automated. TDD and ATDD are adopted widely by the industry because they improve software quality and productivity [21-22].

However, many developers find themselves confused while using TDD and ATDD in their projects, "programmers wanted to know where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails" [3]. Some of the problems of TDD and ATDD are that they are focused on verifying the state of the system rather than the desired behaviour of the system, and that test code is highly coupled with the actual systems' implementation [18, 20]. In addition, in these approaches unstructured and unbounded natural language is used to describe test cases which are hard to understand [3].

BDD is generally regarded as the evolution of TDD and ATDD. BDD is focused on defining fine-grained specifications of the behaviour of the targeting system, in a way that they can be automated. The main goal of BDD is to get executable specifications of a system [3, 20]. BDD relies on ATDD, but in BDD tests are clearly written and easily understandable, because BDD provides a specific ubiquitous language that helps stakeholders to specify their tests. There are also various toolkits supporting BDD, such as JBehave [4], Cucumber [5] and RSpec [6].

Currently, the BDD approach is still under development. The understanding of BDD is far from clear and unanimous. There is no one well-accepted definition of BDD. The descriptions of the characteristics of BDD are vague and scattered. The supporting tools are mainly focused on the implementation phase of the development process, which is a mismatch to BDD's broader coverage of the software development lifecycle. Based on these observations, the main objective of our study is to identify the characteristics of BDD and conceptualize them in an explicit manner, which can serve as a basis for understanding the BDD approach, and for the development and extension of the BDD supporting tools.

This paper is structured as follows. Section II provides a review of the few existing BDD studies. Section III describes the research approach employed in our study. Section IV elaborates on the identified BDD characteristics and presents a conceptual model that encapsulates these characteristics. The last section gives the conclusions and future work.

II. Related Work

There are very few published studies on BDD, most of which take a relatively narrow view of BDD and only treat it as a specific technique of software development. This may be a reflection of the original vision of BDD as a small, simple change from existing TDD practices. Carvalho et al. [8, 9] view BDD as a specification technique that "automatically certifies that all functional requirements are treated properly by source code, through the connection of the textual description of these requirements to automated tests". According to them, BDD starts with textual descriptions of the requirements using specific keywords that tag the type of sentence, indicating how the sentence is going to be treated in the subsequent development phases. Since the focus of their work is on the higher BDD abstraction level, they mainly focus on the set of predetermined tags in BDD that form a simple ubiquitous language. Many details of BDD are not treated in their work. Similarly, Tavares et al. [7] focus on the implication of BDD as a design technique and claim that the aim of BDD is to integrate verification and validation in the design phase in an outside-in style, which implies thinking early on how the client acceptance criteria are before going into the design of each part that composes the functionality. They argue that, as BDD is strongly based on the automation of specification tasks and tests, and it is necessary to have a proper tooling to support it.

Instead, Keogh [10] embraces a broader view of BDD and argues its significance to the whole lifecycle of software development, especially to the business side and the interaction between business and software development. Keogh attempts to unveil the value of BDD using the concepts of Lean thinking, such as value stream, pull, and the PDCA (Plan-DoCheck-Adapt) cycle. In addition, this author argues that BDD permits to deliver value by defining behaviour, and it is focused on learning by encouraging questions, conversations, creative exploration, and feedback. BDD also aids to decouple the learning associated with TDD from the word "test", using the more natural vocabulary of examples and behaviour to elicit requirements and create a shared understanding of the domain. Even though the study in [10] does not provide a comprehensive list of the BDD characteristics, it demonstrates convincingly that BDD has broader implication to software development processes than being just an extension of TDD.

Lazăr et al. [11] also highlight the value of BDD for business domain and the interaction of business and software development, claiming that BDD enables developers and domain experts speak the same language, and encourages collaboration between all project participants. They point out two core principles of BDD: (1) business and technology people should refer to the same system in the same way; and (2) any system should have an identified, verifiable value to the business. Based on this view of BDD, they analyze the BDD approach and present the main BDD concepts as a domain model and a BDD profile. However, their domain model does not allow the specification of business value or the recipient of that value. As a consequence, it is not possible to relate a system or part of it with the business value that it provides, which is inconsistent with the second core BDD principle they claim. Besides, the BDD profile they build does not take into account the relationships among several key concepts of BDD.

III. Research Approach

Based on the objective of our study and the review of related work, the research question we address in our study is: what are the main characteristics of behavior driven development? To this end, the research approach employed in our study is composed of reviewing relevant literature and analysing current BDD toolkits. We started from reviewing the BDD literature. As shown in Section II, the published literature on BDD is very limited. It is difficult to identify the concepts and characteristics of BDD relying on the BDD literature only. To overcome this constraint, we also reviewed the related literature including TDD and Domain Driven Development, since BDD is a combination of a set of concepts from these areas. We used TDD as a baseline to delineate the BDD specific characteristics. That is, what we considered the BDD specific characteristics are those not reported as TDD's.

We analyzed seven current BDD toolkits to verify the BDD characteristics identified from the literature and to discover anyone we missed. There are more than 40 BDD toolkits listed in the BDD Wikipedia page [13] at the moment the toolkits analysis was conducted. To choose the suitable BDD toolkits to study, we used the Wikipedia list as a checklist and consulted one of the BDD mailing lists [23] to decide which BDD toolkits in the list were often used by practitioners. As the result seven most frequently mentioned toolkits in the discussions are included in the analysis. They are: Cucumber [5,18] , Specflow [14], RSpec [6,18], JBehave [4], MSpec [15], StoryQ [12] and NBehave [16]. We grouped JBehave and NBehave under the title of "xBehave Family", and RSpec and MSpec under "xSpec Family", due to the similarities of the toolkits within the same family. Table 1 gives a brief overview of the seven toolkits and the versions that we analysed.

The literature review and toolkits analysis were interwoven steps. After reviewing several studies and drawing up an initial set of the BDD characteristics, we analysed one toolkit at a time using the set of characteristics and recorded how the toolkit supported them. If we found a characteristic in the toolkit that was not in the initial list, we went back to the literature to understand if it could be considered a BDD characteristic, and decided if the initial list should be extended accordingly. This process was repeated for each toolkit.

Table 1: The BDD Toolkits Analysed in Our Study

	xBehave Family		xSpec Family		StoryQ	Cucumber	SpecFlow
	JBehave	NBehave	RSpec	MSpec			
Programming language supported	Java	C#	Ruby	C#	C#	Ruby, Java, Groovy, C#, etc.	C#
Version analysed	3.1.2	0.4.5	2.3	5.1	2.0.4	0.10.0	1.5

IV. Cucumber Model

The last two geeky conversations I had, stumbled upon the same thing – how do you measure the effectiveness of requirements in describing the business to the business and describing the specification to the developer? So, I posed the question "How far away are you from executing your requirements?". If you are going to go through various steps and stages to get to compilation and then execution, then every step is an opportunity for valuable information being lost in translation. If you can compile your requirements immediately then nothing will be lost. Each additional step between requirements description and compilation and execution is an opportunity to confuse the user and the developer and everyone in between. That's why fully dressed use cases are not so

effective as fully dressed behavior driven stories. And that's why BDD is very agile and a great asset in DDD and use cases just don't cut it anymore. Right now, my favorite tool is Cucumber. I can execute the requirements and that raises the clarity ranking of my requirements super high.

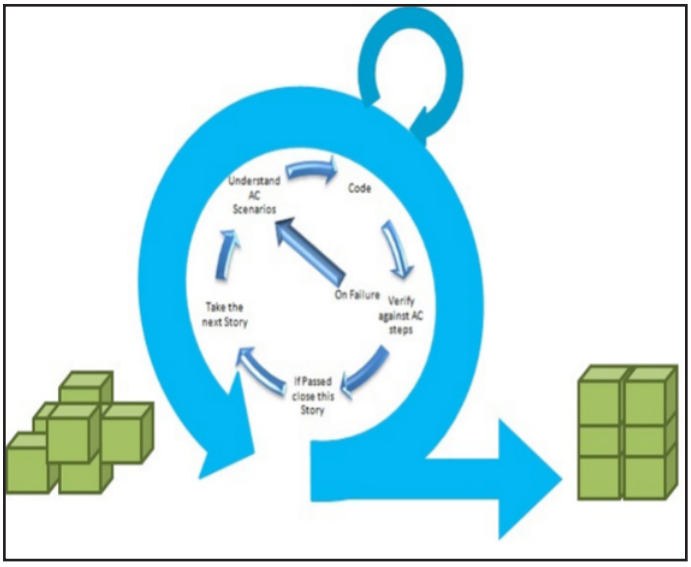

Fig. 1: Sprint Cycle After Adopting ACDD

The activities to be followed in order to build the feature within a sprint cycle by adopting ACDD are listed as follows and as shown in fig. 1.

- The developer assigns the story to self.
- He understands the AC.
- He implements/codes to meet the AC.
- The developer verifies against AC. On failure, he analyzes the reason, fixes the cause, and repeats the process. If the developer is unable to implement any step of the AC within the estimated time, the team can take a decision to create a defect or mark the story as incomplete.
- On success, he moves the story to "done" and starts working on the next story.

V. The Characteristics of BDD

Based on the broad view of BDD we hold which covers the whole range of software development activities, including requirements eliciting, analysis, design and implementation. We have identified six main characteristics of BDD from the literature review and toolkits analysis.

A. Ubiquitous Language

The concept of "ubiquitous language" is at the core of BDD. A ubiquitous language is a language whose structure comes from a domain model. It contains the terms which will be used to Creating a ubiquitous language for a project is crucial since it should be used throughout the development lifecycle. A dictionary is specified at the beginning of the project. Most vocabulary of the ubiquitous language should come from the analysis phase. However, new words can be inserted at any time of the development phases. Creating the ubiquitous language needs to involve anyone (domain experts and developers) who will use the language. In the design and implementation phases, developers will use the language to name classes and methods.

BDD itself also includes a pre-defined simple ubiquitous language for the analysis process, which is domain independent. It is used to structure user stories and scenarios. It will be explained in more detail in the "Plain Text Description with User Story and Scenario Templates" section.

None of the toolkits we analysed supports the creation of a specific ubiquitous language for a project.

B. Iterative Decomposition Process

It is often difficult for developers to find a starting point to communicate with customers during requirements gathering. Customers need some business value to be realized by a software project. Business value is generally difficult to identify and made explicit. Therefore in BDD the analysis starts with identification of the expected behaviours of a system, which are more concrete and easy to identify. The system's behaviours will be derived from the business outcomes it intends to produce. Business outcomes are then drilled down to feature sets. A feature set splits a business outcome into a set of abstract features, which indicate what should be done to achieve the business outcome. Feature sets are derived from discussions between customers and developers on business outcomes. They need to be associated explicitly to the business outcomes they help achieve. Sometimes, one feature set may contain sub feature sets.

Considering that business outcomes are the starting point of BDD process, it is necessary for customers to specify the priority of the business outcomes so that developers know which set of features is more important to be developed first.

A feature is subsequently realised by user stories. User stories provide the context of the features delivered by a system. User stories are user-oriented. User stories describe the interactions between users and a system. There are three questions that should be clarified by a user story:

- What is the role of the user in the user story?
- What feature does the user want?
- What benefit can the user gain if the system provides the feature?

For one user story, there may be different versions in different contexts. The specific instances of a user story are called scenarios. Scenarios should describe specific contexts and outcomes of the user story, which should be provided by customers. Scenarios in BDD are used as acceptance criteria.

The decomposition process described above should be iterative, which implies barely enough up-front analysis. The analysis at one level can stop if it is enough for the implementation even if there are still something potential to be unveiled. None of the toolkits we studied supports the iterative decomposition process.

C. Plain Text Description with User Story and Scenario Templates

In BDD plain text descriptions of features, user stories and scenarios are not in a random format. Pre-defined templates are used in specifying them. The templates are defined using a

simple ubiquitous language that BDD provides. Typically user stories are specified using the following template [3]:

[StoryTitle] (One line describing the story)

As a [Role]

I want a [Feature]

So that I can get [Benefit]

The user story title describes an activity that is done by a user in a given role. The feature provided by the system allows the user to perform the activity, and after performing the activity the user obtains a benefit. Using this template, one can clearly see what feature the system should support and why it should be supported by the system. Developers know which system behaviour they have to implement, and with whom to analyse and discuss the feature. In addition, users have to think if they really need a feature, since they should be able to describe what benefit they will obtain using the feature.

The template for writing scenarios is as below:

Scenario 1: [Scenario Title] Given [Context]

And [Some more contexts]….

When [Event]

Then [Outcome] And [Some more outcomes]…. Scenario2: [Scenario Title] ….

A scenario describes how the system that implements a feature should behave when it is in a specific state and an event happens. The outcome of the scenario is an action that changes the state of the system or produces a system output. We use the term Action instead of System Outcome indicated in [3] because an Action can represent any reactive behavior of the system.

For both, user story and scenario templates, the descriptions in square brackets should be written in the ubiquitous language defined in the project. What's more, they are mapped to tests directly, which means the names of classes and methods should also be written in the ubiquitous language.

The user story templates used in four toolkits we analysed: JBehave, NBehave, SpecFlow and Cucumber, are slightly different than the original one proposed by Dan North. They all have the three elements for defining the role, feature, and benefit of a user story. But they use different words and order.

However, they do not change the semantics and goals of the user story template. Meanwhile, all four toolkits provide a scenario template which follows the structure described previously. In contrast, the xSpec Family and StoryQ do not provide any of the templates since they are focused on the implementation phase only. However, RSpec is usually used together with Cucumber which does provide them.

D. Automated Acceptance Testing with Mapping Rules

BDD inherits the characteristic of automated acceptance testing from ATDD. An acceptance test in BDD is a specification of the behavior of the system, it is an executable specification which verifies the interactions (or behavior) of the objects rather than their states [3, 10].

Developers will start from scenarios produced in one iterative decomposition process. Scenarios will be translated to tests which will drive the implementation. A scenario is composed of several steps. A step is an abstraction that represents one of the elements in a scenario which are: contexts, events, and actions. The meaning of them is: in a particular case of a user story or context C, when event X happens, the answer of the system should be Z. One step is mapped to one test method. In order to pass a scenario, it is necessary to pass all the steps. Each step follows the process of TDD which is "red, green, refactoring" to make it pass.

In BDD all scenarios should be run automatically, which means acceptance criteria should be imported and analysed automatically. The classes implementing the scenarios will read the plain text scenario specifications and execute them. In other words, BDD allows having executable plain text scenarios.

Mapping rules provide a standard for mapping from scenarios to test code (specification code). There are variations of mapping rules in the toolkits we studied. In JBehave, a user story is a file containing a set of scenarios. The name of the file is mapped to a user story class. Each scenario step is mapped to a test method that is located using an annotation describing the step, and usually the test method has the same name as the annotation text. The class containing the step methods does not need to have the name of the scenario.

Cucumber can be integrated with tools like RSpec which allow executing behaviour driven specifications. Cucumber uses regular expressions to perform the mappings. The names of the steps defined in the plain text scenarios should match (using a regular expression) those of the methods in RSpec. In the xSpec Family and StoryQ instead there are no applied mapping rules due to their focus on implementation phase therefore they lack of functionality for analysis.

E. Readable Behaviour Oriented Specification Code

BDD suggests that code should be part of the system's documentation, which is in line with the agile values. Code should be readable, and specification should be part of the code.

The names of methods have to indicate what methods should do. The names of classes and methods should be written in sentences. Code should describe the behaviours of objects. The application of mapping rules help produce readable behaviour oriented code. It ensures that class names and method names be the same as user story titles and scenario titles. Besides, those names should be in the ubiquitous language defined in a project.

StoryQ and the xSpec Family provide APIs that allow developers to specify user stories and scenarios as behaviour driven code. They help structure the code, and make it more readable. JBehave and NBehave also help write scenarios as code and make code readable by means of annotations. SpecFlow generates the scenarios as NUnit tests. In contrast, Cucumber is not focused on the implementation level thus does not support this characteristic.

F. Behaviour Driven at Different Phases

The BDD characteristics we have discussed in the previous sections demonstrate that behaviour driven happens at different phases of software development using the BDD approach. At the initial planning phase, behaviours correspond to business outcomes. At the analysis phase, business outcomes are decomposed into a set of features which capture the behaviour of the targeting system. Besides, behaviour driven is also embodied at the implementation phase. Automated Acceptance testing is an integral part of the implementation in the BDD approach. Testing classes are derived from scenarios and their names follow a set of mapping rules.

The toolkits we analysed do not allow defining business outcomes or features, that is, there is no support to behaviour driven at the planning phase. At the analysis phase, some of them support the definition of user stories and scenarios using the BDD templates. In addition, they provide mapping rules in order to execute the acceptance tests from plain text scenarios. For instance, the xBehave Family, SpecFlow, and RSpec combined with Cucumber, provide such support. In contrast, most of them do permit to write scenarios as code directly, only Cucumber does not.

Table 2: Summarizes the Support of the Seven BDD Toolkits to the Seven BDD Characteristics

Support of the BDD Characteristics		**xBehave Family**		**xSpec Family**		**StoryQ**	**Cucumber**	**SpecFlow**
		JBehave	**NBehave**	**RSpec**	**MSpec**			
Ubiquitous language definition		×	×	×	×	×	×	×
Iterative decomposition process		×	×	×	×	×	×	×
Editing plain text based on	User story template	√	√	×	×	×	√	√
	Scenario template	√	√	×	×	×	√	√
Automated acceptance testing with mapping rules		√	√	×	×	×	√	√
Readable behaviour oriented specification code		√	√	√	√	√	×	√
Behaviour driven at different phases	Planning	×	×	×	×	×	×	×
	Analysis	√	√	×	×	×	√	√
	Implementation	√	√	√	√	√	x	√

Table 2 summarizes the support of the seven BDD toolkits to the seven BDD characteristics. Fig. 1 is a conceptual model, specified as UML class diagram that synthesizes the concepts and relationships presented in the seven BDD characteristics.

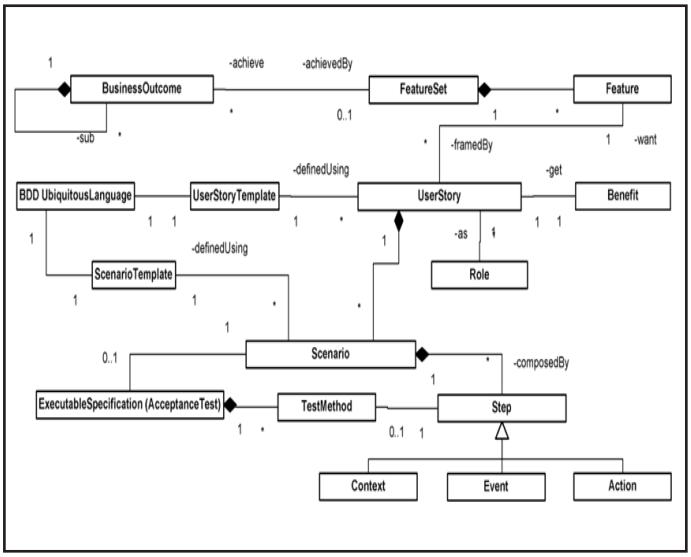

Fig. 2: BDD Conceptual Model

VI. Expected Benefits

Teams already using TDD or ATDD may want to consider BDD for several reasons:

1. BDD offers more precise guidance on organizing the conversation between developers. Testers and domain experts.
2. Notations originating in the BDD approach, in particular the given-when-then canvas, are closer to everyday language and have a shallower learning curve compared to those of tools such as Fit/Fitness.
3. Tools targeting a BDD approach generally afford the automatic generation of technical and end user documentation from BDD "specifications".

VII. Conclusion

BDD is a combination of several approaches, including ubiquitous language, TDD and automated acceptance testing. It optimizes the connections of these approaches to make the most out of each single approach. In this study we identified six BDD characteristics through literature review and toolkits analysis. Our study shows that these characteristics are interlinked. Therefore these characteristics should be embraced in a holistic way in a software development project to get the full potential benefits of the BDD approach. We also find that the BDD toolkits studied mainly focused on the implementation phase of a software project and provide limited support to the analysis phase, and none to the planning phase. We also presented a conceptual model of BDD based on the results of our study, to provide a more explicit and formal description of the BDD concepts and their relationships.

The results of our study indicate several potential venues for future research. Our study shows that most of the toolkits lack the support of the BDD characteristics related to the planning and analysis phases. Therefore one future study could extend an existing BDD toolkit or develop a new one based on the proposed conceptual model. The new toolkit will provide support to the software development activities that need collaboration between business and development team. Another future study could extend and implement additional mapping rules. The existing mapping rules in the BDD toolkits only map user stories and scenarios to code. Feature sets might be mapped to namespaces or packages too, where the test classes of a scenario can be located.

References

[1] K. Beck,"Test-Driven Development", By Example. Addison Wesley, 2003.
[2] L. Koskela,"Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007.
[3] D. North,"Introducing BDD, (2006). [Online] Available: http://dannorth.net/introducing-bdd [Accessed December 13, 2010].
[4] JBehave, [Online] Available: http://jbehave.org/ [Accessed December 13, 2010]
[5] Cucumber, [Online] Available: http://cukes.info/ [Accessed December 13, 2010]
[6] RSpec, [Online] Available: http://rspec.info/ [Accessed December 13, 2010]
[7] H.P. Tavares, G. GuimarãesRezende, V. Mota, R. SoaresManhães, R., R. Atem De Carvalho,"A tool stack for implementing Behaviour Driven Development in Python Language", CoRR, 2010.
[8] R. Carvalho, R. SoaresManhães, F.L. de Carvalho, "Filling the Gap between Business Process Modeling and Behavior Driven Development", CoRR, 2008.
[9] R. Carvalho, F.L. De Carvalho, R. Soares,"Mapping Business Process Modeling constructs to Behavior Driven Development Ubiquitous Language", CoRR, 2010.
[10] E. Keogh,"BDD: A Lean Toolkit", In Processings of Lean Software & Systems Conference, Atlanta, 2010.

Dr.Ravi Saripalle Working As Director, Center for Innovation GVP College of Engineering (A), Visakhapatnam.

Buddharaju Shanmukh Varma from GVP College of Engineering (A), Visakhapatnam. Areas of research interest include Programming research.

Raghukanth Reddy Gudimetla from GVP College of Engineering (A), Visakhapatnam. .Areas of research interest include Programming research.

Sudeepa Gorle from GVP College of Engineering (A), Visakhapatnam. Areas of interests are Programming research.