Towards a Taxonomy for Applying Behavior-Driven Development (BDD)

David Faragó[1], Mario Friske[2], Dehla Sokenou[3]

[1]QPR Technologies, Karlsruhe [2]Friske Consulting, Berlin [3]GEBIT Solutions GmbH, Berlin

Abstract. Behavior-Driven Development (BDD) is a topic currently much talked about, especially in the agile community. Small scale examples of BDD suggest an intuitive and easy use, but experience shows that in practice, especially large projects, its application becomes elaborate and challenging. This paints an inconsistent picture about BDD. So, what are the requirements for a successful application of BDD?

We have identified, discussed, and classified the core aspects of applying BDD. Depending on the application context, an aspect can speak for or against the use of BDD. These aspects and their pro and contra arguments are this article's main contribution. Everyone can use these aspects to decide whether and how to use BDD in their individual project context.

1 What is BDD today

BDD is a software development method derived from Test-Driven Development (TDD). It is an outside-in software agile development methodology: it focuses on satisfying the needs of stakeholders by using user scenarios as starting point for the implementation. Furthermore, it provides a ubiquitous specification language, which guarantees that all stakeholders can communicate with each other – avoiding the frequent misunderstandings between domain experts and developers. Finally, by making the specifications semi-formal and directly executable, tools can check the specified scenarios automatically, thereby promoting continuous validation and unambiguousness of specifications.

A widely accepted specification language is Gherkin, with Cucumber as its corresponding BDD tooling. In cases when BDD is used for testing purposes only, the method is sometimes called Behavior-Driven Testing (BDT) to draw a distinction.

2 Discussions on BDD within our AK

Our working group "Testing of Object-Oriented Programs / Model-based Testing" (AK TOOP) is part of the GI's professional group "Test, Analysis and Verification" (TAV). Usually, we meet in person during TAV events and continuously work on interesting topics between meetings using online collaboration tools.

In our last group meetings, we focused on the relation between Model-based Testing (MBT), Keyword-Driven-Testing (KDT) and Behavior-Driven Development (BDD), see [2] and [3]. These discussions identified some questions and aspects about introducing and running BDD that need further clarification.

Aspects of BDD have already been discussed elsewhere: [9] lists six aspects that are relevant but very general and mainly cover specifications and processes. [5] goes more into detail and touches the technical application and the ROI of BDD, but does not cover all aspects we investigate, does not structure the aspects clearly.

An empirical study in [8] lists ten quality attributes as relevant for BDD scenarios: concise, estimable, feasible, negotiable, prioritized, small, testable, understandable, unambiguous, and valuable. These attributes are from other types of requirement specifications (like use cases or user stories) and are all covered by our aspects. The literature review [4] isolated six major aspects of BDD, which we also cover: ubiquitous language, requirements, acceptance tests, tools, collaboration, and automation.

In our discussions, we focused on

1. aspects relevant for applying state-of-the-art BDD in practice, with argument for and against BDD,
2. completeness of these aspects, covering all aspects listed in [9] and [5],
3. a classification of these aspects, leading towards a taxonomy of BDD.

In the following, we summarize the results.

3 Sixteen Aspects of BDD

In our discussions, we came across many situations, aspects and arguments for and against BDD. From these discussions, we isolated 4 x 4 aspects that we grouped into 4 categories: specification, process, technical application and return on investment (ROI). For each aspect, we collected arguments for and against

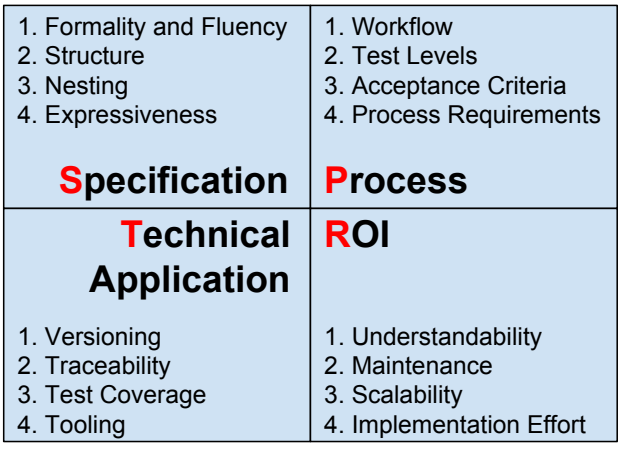

Figure 1: Relevant aspects of applying BDD

BDD. Hence this paper focuses on establishing the aspects in the table on the right as a basis for future discussions and investigations about BDD – for establishing a taxonomy of BDD.

3.1 Specification

3.1.1 S1 Formality and Fluency

In BDD, requirements are captured using a scenario specification language like the popular Gherkin. Such languages are based on natural language but define a few special keywords like "Given", "When" and "Then".

Gherkin statements can be written in such a way that they are like natural language, even when using parameters. The range can vary from non-reusable statements copied without any change from an already existing specification up to parameterized statements, explicitly designed for reusability, but still conforming to the grammar of the underlying natural language.

BDD scenarios innately focus on domain level not on implementation level. This has the advantage that test cases are instantly readable and understandable by subject matter experts. However, developers and test automation engineers need getting used to an example-oriented style of specification where statements are not primarily designed for reuse.

3.1.2 S2 Structure

A common question in software projects is how to structure test automation. For instance, the popular JUnit does not come with out-of-the-box structuring guidelines. Gherkin/Cucumber fills a gap by forcing the user to define a set of reusable steps on business level, which is usually is not explicitly addressed when implementing classical unit-based tests.

But it remains open how to organize the underlying automation layer. Some approaches suggest using three layers of abstraction: the business rules, the system workflow, and the specific user activities.

Providing an additional business layer is a clear advantage of the BDD methodology but can be challenging for developers to transform a set of self-contained scenarios that follow the Specification by Example paradigm [1] into a flexible structure of underlying parametrized and re-usable implementation functions.

3.1.3 S3 Nesting

Different to other specification techniques, BDD only supports a small set of expressions. The strict form of BDD statements (given, when, then) increases the readability and comprehensibility of the specification but reduces the expressiveness (see S4).

BDD is missing case distinction and repeated statements as well as the possibility to modularize the scenarios using sub-scenarios. Thus, each scenario is self-contained but hides the potentially existing complexity on the implementation level. Consider e.g. the login process which may be defined as a simple BDD statement ("given a logged-in user") but is implemented in the specified system as a complex workflow and also the BDD automation consists of a series of steps to validate the logged-in user.

Due to the lack of modularization techniques and control-flow instructions (see S1 and S2), the right level of abstraction for scenarios and comprised steps must be chosen to avoid unnecessary complexity by keeping scenarios small and concise without listing any irrelevant details. This is one of the challenges when applying BDD.

3.1.4 S4 Expressiveness

The previous aspects result in lower expressiveness of BDD compared to code-based or model-based tests, which means you need to write more tests or more complex tests, or you will find fewer bugs. So lower expressiveness of the specification language directly leads to a test suite that has a lower fault-finding effectiveness. However, since BDD's intention is to focus on and specify only certain examples, its lower expressiveness is suitable if you keep the lower fault-finding effectiveness in mind.

Since BDD is a form of specification-by-example, all pros and cons of this approach also apply to it. For example, the primary focus on significant cases is one of the major advantages of specification-by-example and therefore BDD, but may lead to incompleteness of the specification, be it caused by missing scenarios for relevant cases or caused by a wrong interpolation from specified to similar cases. A guidance to avoid typical faults in example-based specifications like the lack of completeness can be found in [1].

3.2 Process

3.2.1 P1 Workflow

The most prominent BDD workflow comprises a language (Gherkin), a tool (Cucumber) and a process (see P4 and [9]). The process relies on the outside-in-development workflow. Cucumber natively supports this workflow on the testing side: First business analysts write scenarios in Gherkin. The quality of the scenarios strongly influences how efficient and effective the remaining workflow is, for instance scenarios should be on the right level of abstraction (see S3), prioritized, and estimable. These properties increase agility and negotiability (see P4 and [8]). After specifying the scenarios, Cucumber generates a code stub plus a corresponding matching clause for each missing step implementation. Without any change, generated code fragments can serve to set up frames for missing step implementations rapidly. Thereafter, the pattern matcher will no longer fail, but will point to remaining implementation tasks by throwing "missing step implementation exceptions".

On the development side, the workflow is not as defined as on the testing side. In principle BDD allows to realize Acceptance Test Driven Development (ATDD). ATDD aims to shift TDD's unit-test-level-centric Red-Green-Refactor-Cycle to a higher-level Specify-Develop-Deliver-Cycle. In practice, especially in large-scale projects, it is very challenging to implement such an approach. The result is that BDD is often used for testing purposes only, resulting in BDT.

3.2.2 P2 Test Levels

BDD as a specification language defines scenarios on a user-centered view. Thus, naturally these scenarios are situated on acceptance test level when using the defined scenarios as input for tests (see P3). Of course, BDD and especially Gherkin as language are not limited to acceptance tests (see also [9]): Tests on other levels may also be described using BDD techniques, including unit, integration, system tests and system integration test. For instance, Cucumber's ability to specify data-driven tests can be used to realize unit tests on a very detailed level.

On the one side establishing BDD techniques across different test levels is a great step towards methodical and technological convergence. On the other side in our opinion BDD is not the perfect choice for all required testing activities. Classical examples for test levels that require approaches beyond Specification by Example (SbE) [1] are performance and stress tests.

Another aspect when applying BDD across different test levels is the achieved test coverage (see T3).

3.2.3 P3 Acceptance Criteria

One of the main strengths of BDD is its focus on acceptance criteria. This is no surprise since BDD emerged from ATDD by offering domain specific and ubiquitous specifications (see S1), enabling domain experts and developers to collaboratively derive acceptance criteria formulated in the domain language (see also [9]). Furthermore, these story-based specifications should be derived outside-in, i.e. with business value in mind, yielding value-based acceptance criteria. Finally, these specifications are executable as test cases, avoiding any gap between the specification and the implementation and providing continuous validation to guarantee that a test will fail the moment the implementation no longer conforms to the acceptance criteria. For these reasons, BDD has evolved into a popular practice from agile requirements engineering and design down to software development and testing.

However, some of BDD's deficits require the use of other methods for specifying and testing in certain situations, e.g. when: semi-formalness (see S1) and low expressiveness (see S4) cause the need for stronger functional specifications, e.g. specification-based BDD, a form of property-based testing; the outside-in approach might tempt you to focus only on the happy paths, so you can easily miss special cases, e.g. exceptional situations; non-functional requirements must be tested automatically; for instance, robustness and security tests are often covered better by fuzzing or MBT; high test coverage must be guaranteed (see T3 below), where methods like MBT shine certifications and standards are relevant, e.g. ISO 26262 requiring more formal and rigorous RE and quality assurance methods.

Having multiple specifications or testing methods causes the usual disadvantages from redundancies, especially double work, risk of inconsistencies and the need for traceability between them (see T2).

3.2.4 P4 Process Requirements

In each project that is going to apply BDD, some typical questions need to be addressed: Who writes and modifies the feature files? Who programs the step implementations? Do we need different libraries for module tests, integration tests and product tests? How and by whom will different contributions be integrated into a "common library"? Are there any "owners" of common libraries? How are shared steps identified? Are there guidelines for writing scenarios and steps? How does the project deal with changes? Answering these questions might point out that existing roles, workflows and procedures require changes. Before introducing BDD, each project should determine if it can handle all required changes.

3.3 Technical Application

3.3.1 T1 Versioning

As BDD is text-based, it is easy to archive scenarios using a version control system and to compare current and older versions of a BDD scenario. Comparisons are much easier for text-based artefacts than for models, even if the models have a textual representation – a text-based comparison for models is less helpful. Note that the comparison mechanism only works on files that preserve (beyond others) the scenario order.

Additionally, the code base for the BDD project can be archived like any other code.

3.3.2 T2 Traceability

Traceability is the ability to link requirements to other artifacts like code, test cases, or bugs. In software testing, traceability allows to trace back to the corresponding requirements, for instance if a test fails, or if you want to compute the requirements coverage (see T3) of a test suite.

Using BDD's executable specifications, defined scenarios can be run as acceptance tests without any change. BDD scenarios serve simultaneously as acceptance criteria and tests (see P3). Consequently,

BDD simplifies traceability between user stories, related scenarios and test runs, i.e. executed scenarios. BDD's specifications do not directly enable traceability to code, but by applying the ATDD workflow (see P1) scenario-related code changes can be traced.

3.3.3 T3 Test Coverage

A common misunderstanding is that BDD's main purpose is test automation. This also leads to the attempt to achieve high test coverage by using BDD. The term BDT is often used by those having this misunderstanding, as well as those trying to clarify this misunderstanding.

Looking at the other aspects reveals that BDD's purpose is not test automation, but application on the requirements and process level. That is why [1] discourages doing combinatorial testing using BDD and advises to pick examples to improve understanding: "There isn't much point in going through examples that illustrate existing cases; that doesn't improve understanding. When illustrating using examples, look for examples that move the discussion forward and improve understanding." In short: BDD tests "aren't replacements for combinatorial regression testing." Instead, BDD tests should focus on the functionality leading the business to success: "Decide what to cover and what not to cover depending on the conditions of success for the story." [1]

Thus, other methods are more suitable to achieve high test coverage, e.g. MBT and fuzzing (see P3 and R4). To purely measure test coverage, however, BDD is sufficient, even for requirements coverage (see T2).

3.3.4 T4 Tooling

BDD is not usable without a good tool support because specifications should be made executable. BDD tools offer an easy way to bind a scenario specification and the corresponding test implementation and to detect a missing step implementation.

Many tools are available: from BDD support for numerous programming languages (e.g. Cucumber, JBehave, SpecFlow, Jasmine), execution environments, to integrations in IDEs (e.g. Eclipse, Visual Studio) and even sole test tools (e.g. TestLeft).

As scenarios may grow and become complex and have semantic similarities (e.g. different ways to describe a logged-in user), support for refactoring of scenarios is needed but only partly provided by tools, e.g. the tool Hiptest supports renaming and the replacement of some steps by an action and vice versa, but more complex refactoring operations are not provided. Practically each project needs to verify that there is a working combination of existing testing code, deployed IDEs and compatible plugins providing support for the specific version of the desired BDD language.

3.4 ROI

3.4.1 R1 Understandability

To productively apply BDD, many parts need to be understood: the concept, the workflow, the tool and specification language being applied, and finally the concrete tests being specified.

As described in the introduction, the core concept is easy to grasp, as is the workflow (see P1). This is even more so if you are familiar with TDD – after all, the original and technical goal of BDD was "TDD done right" (see [4]).

Aspect T4 explained that many tools are open source and have a big community. Thus, there is a lot of online information, but the availability of thorough manuals is limited. In summary, the aid in getting started with a specific BDD tool is usually much higher than in diving deeper into the subject, to apply BDD in the right way for a large project, to cope with maintenance and scalability issues (see R2, R3).

Due to the semi-formality (see S1), the simple structure (see S2), the avoidance of nesting (see S3), the Gherkin language is very simple, making it easy to understand. But the low expressiveness spawned alternative specification languages (see S4) that are more difficult, as is the decision which specification language to pick when. Having many test cases, textual languages are usually simpler to understand than graphical specifications and more suitable for test cases that are non-nested, flat, singular examples. Textual searching, diffing and versioning also helps in understanding and researching within the tests. However, occasionally graphical test specifications (e.g. in BPMN) can be preferable even within BDD [7].

Gherkin leads to very understandable (story-based) tests, for developers as well as managers and domain experts. This ubiquitous specification language is one of the key selling points of BDD.

3.4.2 R2 Maintenance

The lack of expressiveness of the Gherkin languages leads to scenarios that are difficult to maintain - the longer a scenario is, the more complex is the content of each of the Gherkin sections (given, when, then). There are two way to deal with complexity. The first is to abstract from the concrete steps, e.g. if a complex login scenario is reduced to a short precondition ("given the user is logged in") – the complexity will then be implemented on the code's side. The second way is to describe the complete scenario in Gherkin, having more mapping in code but a lot of statements in the specification. Both ways may be mixed.

Both ways are difficult to maintain. Assume a change in the login process. In the first case, the implementation has to be changed which may have an impact on other scenarios which can be linked to the same code. In the second case, all affected scenarios

have to be changed. Mixing both ways, that is to build logical units for parts of a scenario which are implemented as functions in code, may increase the maintainability of BDD scenarios.

Maintenance for BDD can explode when you have thousands of BDD scenarios, traceability to user stories, test automation, and impact analyses. Reusability is often very difficult in BDD as it does not enforce modularization (see S2, S3, S4).

3.4.3 R3 Scalability

Does BDD scale well, meaning that it can be applied effectively even if the size of the team, source code, set of acceptance criteria and complexity of the application as well as domain rises? The expressiveness of the specification (see S4), weak test coverage (see T3) and maintenance (see R2) can break BDD's feasibility and effectiveness if it is not applied masterfully.

The SbE book [1] lists some example large scale projects that have successfully applied BDD. These project reports point out that most successful teams were very competent and/or had help from very strong testing and BDD consultants, ensuring that BDD was applied masterfully.

3.4.4 R4 Implementation Effort

The efforts for introducing and implementing BDD in a team can be partitioned into the initial costs until you get BDD running, and thereafter the running costs. The previous aspects have shown that BDD's initial costs are relatively low compared to other methods (such as model-based or keyword-driven testing, see [2]), mainly because understanding the concept, the workflow, the tool and specification language is relatively easy. However, the running costs can become huge, mainly due to maintenance and scalability issues (see aspect R2).

4 Summary

We have listed and categorized all aspects we find relevant for applying BDD in practice. They serve as a basis for future discussions and investigations about BDD and are thus a step towards a taxonomy.

In our own discussions, these aspects and their categorization have shown to be particularly useful to assess the potential application of BDD in specific projects and compare its benefits to alternative methods - or investigate combinations of methods.

For instance, our section about implementation effort (see R4) comprises the efforts investigated in all other aspects and shows that BDD has very low initial and very high running costs – reciprocal to, for instance, MBT. This might be a reason why BDD is a hype but not often found in large projects, as our discussions have shown. In contrast, it has been shown that MBT can be applied very successfully in large projects (see e.g. [6]), but gets adopted rarely.

This finding is also substantiated by the success projects reported in [1]: Almost all larger projects that do apply BDD had consulting from BDD experts about its effective application. Furthermore, one of the projects showed that it can take twice as long to go from solid TDD to solid, beneficial BDD than it took to go from weak engineering to solid TDD.

We plan to continue our discussions and believe these 4 x 4 aspects are a solid foundation that can help many to evaluate BDD in their context and is thus a step towards deriving an established taxonomy for applying BDD – similar to [10] for MBT approaches.

5 Acknowledgments

We thank Karsten Dörges, Benedikt Eberhardinger, Dierk Ehmke, Matthias Hamburg, Jan Giesen, Karoly Kiss, Andrej Pietschker, and Andreas Spillner for the productive discussions during the last group meetings of our AK TOOP in Munich and Bremerhaven. This article is based on the results of these meetings.

References

[1] G. Adzic. *Specification by Example: How Successful Teams Deliver the Right Software (1st ed.).* Manning Publications Co., 2011.

[2] C. Brandes, B. Eberhardinger, D. Faragó, M. Friske, B. Güldali, A. Pietschker. *Drei Methoden, ein Ziel: Testautomatisierung mit BDD, MBT und KDT im Vergleich.* STT 35, Heft 3, 2015.

[3] B. Eberhardinger, D. Faragó, M. Friske, D. Sokenou. *Aktuelle Fragestellungen zum Zusammenspiel von BDD, MBT und KDT.* STT 36, Heft 3, 2016.

[4] A. Egbreghts. *A Literature Review of Behavior Driven Development using Grounded Theory.* 27th Twente Student Conference on IT, 2017.

[5] B. Holz. *Increase Collaboration and Drive Agility With Behavior-Driven Development.* Gartner, 2015.

[6] W. Grieskamp, N. Kicillof, K. Stobie, V. Braberman. *Model-based Quality Assurance of Protocol Documentation: Tools and Methodology.* STVR 21.1, 2011.

[7] D. Lübke, T. van Lessen. *Modeling Test Cases in BPMN for Behavior-Driven Development.* IEEE Software. 33 (5): 15–21, 2016.

[8] G. Oliveira, S. Marczak. *On the Empirical Evaluation of BDD Scenarios Quality: Preliminary Findings of an Empirical Study.* IEEE 25th International RE Conference Workshops, 2017.

[9] C. Solís, X. Wang. *A Study of the Characteristics of Behaviour Driven Development.* 37th EUROMICRO Conference on SE and Advanced Applications. 2011.

[10] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann 2007.