PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

QUALITY INSPECTION TO EVALUATE BDD SCENARIOS

GABRIEL P.A DE OLIVEIRA

Research plan submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Sabrina Marczak

Porto Alegre
2016

1. INTRODUCTION

Behavior-Driven Development (**BDD**) is an agile practice which uses an ubiquitous language, one that business people and technical people can understand, to describe and model a system [Sma14]. The model is formed by a series of textual scenarios, expressed in a format known as Gherkin, designed to be both easily understandable for all stakeholders and easy to automate using dedicated tools.

Writers of BDD scenarios acting on software development teams does not have a standard set of rules to educate themselves on what a "good" BDD scenario is. They can only compare their work with a few guidelines and examples of "good" and "bad" scenarios found on [Sma14] and other informal internet references. However, this comparison is often misguided, as the writer's application context is hardly compared to the book examples context, and potentially incomplete, as the few guidelines are not a set of enforcement rules as those existent for use cases on [PAJ+11].

[MST01] said that many people have had the experience of writing a piece of text that is felt to be of high quality, only to read it over later and be surprised to discover grammatical, orthographic, and concordance mistakes as well as ideas that could simply be expressed in a better way. The authors state that the reasons for reviewing software deliverables, such as source code, project design or requirements specifications should, are analogous to those for reviewing written text. [Lai02] highlight that it has been claimed that inspection technologies can lead to the detection and correction of anywhere between 50 percent and 90 percent of the defects.

Even with that mentioned advantage, on the best of our knowledge, there is no study addressing the problem of what makes a "good" BDD scenario, the definition of quality on BDD scenarios or how to inspect it properly. However, there are studies with similar goals but focused on other requirements format. For instance, use cases quality is discussed on [Coc00], who also provides guidelines on how to write and a questionnaire on how to validate use cases, as well as on [PAJ+11], where the authors propose a set of rules and guidelines to textual use cases writing along with desirable qualities of use cases that should be seen when expressing requirements, specification and design. [EM12] performs a systematic review in order to present a summary on use cases quality attributes and guidelines examples to evaluate them. User stories are another requirements format there has been explored under the lens of quality attributes. [LDVB15] created the QUS framework to evaluate user stories according to syntactic, semantic and pragmatic criteria built and the AQUSA application that implemented that framework. Generic scenarios and generic examples, both without a given explicitly defined format, were also explored by [AM04] and [Adz11], respectively. Finally, examples of "good" and "bad" requirements are shown when discussing user stories by [LDVB15] in a similar manner as those examples for BDD scenarios shown by [Sma14].

Other studies are concerned with broader categories in a more specific context, like [Dun01], who focus on agile methodologies requirements, and [HZ15], who focus on just-in-time requirements. On the later, the need to investigate further on how to generate BDD scenarios with quality is explicitly expressed. However, no known work have been published on that direction so far.

While the majority of those studies has only briefly approached the matter of requirements textual writing quality, others have succeeded on generating quality guidelines that would help requirements writers to create better artifacts. Use cases quality guidelines have been show on [Coc00] and [PAJ+11], while guidelines for generic scenarios were seen on [AM04].

Still, there are no detailed analysis like those for BDD scenarios. Due to the lack of appropriate studies on BDD scenarios quality, this research plans to dwells on it. It aims to create an inspection reading technique, based on the academic known quality attributes for requirements and practitioners opinions, that would proper serve as guidance to software development teams who want to assess the quality of their BDD scenarios.

The remainder of this paper is organized as follows: chapter 2 explains why this study is important; chapter 3 describes the objectives that are meant to be fulfilled by the end of the study; chapter 4 summarizes the theoretical background on user stories, acceptance tests and requirements quality;chapter 5 details the research methodology to be followed on the future research and how the future activities are organized; chapter 6 describes the current progress on the research plan activities and an estimated timetable to fulfill them.

2. RESEARCH MOTIVATION

The opening words of Cockburn on his book about use-cases [Coc00] ring true for BDD as well. Writing behavioral requirements for software systems seems easy enough - just write about how to use the system and try to sound like the examples shown on [Sma14]. Faced with writing, one suddenly comes face to face with the question, "Exactly what am I supposed to write - how much, how little, what details?" That turns out to be a difficult question to answer. The problem is that writing BDD scenarios (and requirements, in general) is fundamentally an exercise in writing prose essays, with all the difficulties in articulating good that comes with prose writing in general. In addition, the writer is faced with the knowledge that the complete, accurate and concise documenting of requirements is of vital, perhaps paramount importance within software development, and errors made in this phase are often considered the most difficult to solve and most costly to fix [PAJ+11].

Requirements validation is a traditional requirements engineering process phase known to support the three other activities (requirements elicitation, requirements analysis and requirements specification) by identifying and correcting errors in the requirements, as described by [HDLP15]. While quality is ultimately determined by the needs of the stakeholders who will use the requirements or the designs, acceptable quality requirements exhibit many of the characteristics described on the the Business Analyst Body of Knowledge [IIB09] [IIB15].

According to [IIB15], reviews can be used to inspect requirements documentation to identify requirements that are not of acceptable quality.

The Agile Manifesto [KB+01] values individuals and interactions much more than processes and tools. On agile contexts, lengthy documentations are not as important as working software, implying that the time spent on requirements validation is diminished due to the focus on communication and collaboration. As reported by [HDLP15], agile requirements engineering is more flexible and reactive than a traditional, incremental approach. This thought is reinforced by [PEM03], who shows that requirements validation on agile contexts is focused on frequent review meetings and acceptance tests.

The lack of documentation is mentioned to potentially cause long-term problems for agile teams, such as knowledge loss improvement when team members become unavailable and lack of training material to new members [PEM03]. The same problem of minimal documentation is also noted by one of the companies interviewed by [CR08], that reported that it may cause a variety of problems, such as inability to scale the software, evolve the application over time, and induct new members into the development team.

In order to mitigate the before-mentioned problems, the written documentation generated on agile projects need to be of "good" quality. One common practice of Requirement Enginnering (RE) on these contexts, as shown by [CR08], are Acceptance Tests. They are

written to express many of the details that result from the conversations between customers and developers, as mentioned by [Coh04], typically whenever the customer and developers talk about the story and want to capture explicit details or as part of a dedicated effort at the start of an iteration but before programming begins or even whenever new tests are discovered during or after the programming of the story. [HS12] has shown how the use of automated acceptance test-driven development can be seen as a mix of the traditional RE focus on documentation and the agile focus on iterative communication. According to [Gar12], BDD scenarios are a common format of acceptance tests.

However, there wasn't any mention to review meetings centered on the validation of them. Thus, there is still the need to define a review technique that suits well the communication-focused iterative agile RE while also helping the software development team to validate their own work. Due to the fact that it is not that hard to say what a "good" BDD scenario looks like, as [Sma14] have written good and bad examples of BDD scenarios on his book, we will focus on that ATDD format.

In summary, what we wish to accomplish in the current study is to proper describe a BDD scenario based on quality attributes similar to those found on [IIB09] and [IIB15], so a scenario writer can have a review technique to validate it properly.

3. RESEARCH OBJECTIVES

[Cre08] described the purpose statement and the last part of it says that a researcher should describe what is intended to be accomplish. Therefore, this chapter uses the motivation taken from Chapter 2 as a guide to frame the research goal, research questions and objectives of this study.

3.1 Main Research Goal and Questions

The main objective of this research is to create a new inspection reading technique or adapt an existing one to assess the quality of BDD scenarios. To accomplish that, some research questions will be used to drive the research goal:

Research Question 1 (RQ1):

What is a "good" BDD scenario, in terms of the quality attributes it demonstrates, for a member of a software development team ?

Research Question 2 (RQ2):

How are the quality attributes used during the evaluation of a single BDD scenario by a member from a software development team ?

3.2 Objectives

This research have the following objectives:

- Summarize the existent quality attributes applicable to acceptance tests, on the BDD scenario format;
- Summarize the problems that the written form of BDD scenarios may have, according to the experience of its practitioners;
- Propose a new inspection reading technique or adapt an existing one that uses the quality attributes to guide reviewers on their evaluation of BDD scenarios;
- Validate the effectiveness and usefulness of the proposed reading technique with practitioners;

4. THEORETICAL BACKGROUND

In order to study BDD scenarios quality we first need to understand where requirements quality came from and how to evaluate it. This chapter introduces some concepts around those topics in order to answer some informal questions that comes prior to our research topic: what requirements formats already have solid quality definitions, what are the most common evaluation techniques, what agile requirements representations exists, and how BDD is connected with it all.

4.1 Traditional Requirements

A requirement, according to [IIB09], is either a condition or capacity necessary to solve a problem or reach a goal for an interested party or some characteristic that a solution or component should possess or acquire in order to fulfill some form of contract. Additionally, a requirement is also a written representation of this condition or capacity, or a usable representation of a need focused on understanding what kind of value could be delivered to a client if a requirement is fulfilled, according to [IIB15].

When classified according to their purpose, they can be called **business requirements**, **stakeholder requirements** and **solution requirements**. The first are statements of goals, objectives, and outcomes that describe why a change has been initiated, while the second are the needs of stakeholders that must be met in order to fulfill business requirements. The later describe the capabilities and qualities of a solution and provide the appropriate level of details to allow the proper implementation of a solution and can be divided into functional requirements (that describes the capabilities a solution must have in terms of the behaviour and information to manage) and non-functional requirements (that describes conditions under which a solution must remain effective).

4.1.1 Use Cases

According to [Coc00], use cases capture a contract between the stakeholders of a system about its behavior and describes the system's behavior under various conditions by interacting with one of the stakeholders (the *primary actor*, who want to perform an action and achieve a certain goal). Different sequences of behavior, or scenarios, can unfold, and the use case collects together those different scenarios. They're used to express behavioral requirements for software systems and can be put into service to stimulate discussion within a team about an upcoming system. They might later use that the use case form to document

the actual requirements. Besides the primary actor, that interacts with the system, a use case has other parts as well: the *scope* identifies the system that we are discussing, the *preconditions* and *guarantees* say what must be true before and after the use case runs, the *main success scenario* is a case in which nothing goes wrong and the *extensions section* describes what can happen differently during that scenario.

4.1.2 Traditional Requirements Quality

Requirements validation is a phase on traditional requirements engineering process that is known to support the three other activities (requirements elicitation, requirements analysis and requirements specification) by identifying and correcting errors in the requirements [HDLP15]. According to [GFL+13], quality indicators must not provide numerical evaluations only, but first of all they must point out concrete defects and provide suggestions for improvement, just like a spell and grammar checker can help to improve the quality of a text. For [DOJ+93] a perfect software requirements specification is impossible, as some qualities may be achieved only on the expense of others. The author also implies that one must be careful to recognize that although quality is attainable, perfection is not.

The Business Analyst Body of Knowledge says that, while quality is ultimately determined by the needs of the stakeholders who will use the requirements or the designs, acceptable quality requirements exhibit many characteristics. The second edition [IIB09] describes eight characteristics a requirement must have in order to be a quality one, as follows: cohesion, completeness, consistency, correction, viability, adaptability, unambiguity and testability. The third edition [IIB15] bring nine: atomic, complete, consistent, concise, feasible, unambiguous, testable, prioritized and understandable. Both editions define what each characteristic means, but does not provide any measurement guidance.

[GFL+13] list other 11 properties along with analytical metrics that would later help the authors to build a quality framework and implement it on the requirements quality analyzer tool. The characteristics are as follows: Atomicity, Precision, Completeness, Consistency, Understandability, Unambiguity, Traceability, Abstraction, Validability, Verifiability, Modifiability. The measurable indicators are: Size, Readability, Punctuation, Acron. and abbrev., Connective terms, Imprecise terms, Design terms, Imperative verbs, Conditional verbs, Passive voice, Domain terms, Versions, Nesting, Dependencies, Overlappings.

The measurement formalism concern is also present on [DOJ+93], to whom a quality software requirements specification is one that contributes to successfully, cost-effective creation of software that solver real user needs and exhibits 24 quality attributes: Unambiguous, Complete , Correct, Understandable, Verifiable, Internally Consistent, Externally Consistent, Achievable, Concise , Design Independent, Traceable, Modifiable, Electronically Stored, Executable/Interpretable, Annotated by Relative Importance, Annotated by Relative

Stability, Annotated by Version, Not Redundant, At Right Level of Detail, Precise, Reusable , Traced, Organized, Cross-Referenced. His work also define each attribute, provide ideas on measuring them, provide a recommendation of weight relative to other attributes and describe types of activities that can be used to optimize the present of each.

Use cases quality is discussed in details by [PAJ+11], that summarizes prior works on that area (such as the rules found by [Coc00]) and proposes refined rules based on discourse process theory, such as avoiding the use of pronouns, use active voice over passive one, achieve simplicity trough avoiding to use negative forms, adjectives and adverbs and use of discourse cues and the effect of readers background and goals. Also, desirable quality attributes of use cases are listed, that may be suited for certain project phases but not others as follows: standard format, completeness, conciseness, accuracy, logic, coherence, appropriate level of detail, consistent level of abstraction, readability, use of natural language and embellishment. The authors work create rules, that should be enforced and must be obeyed, and guidelines, which indicate an ideal that cannot always be followed, that could best produce those attributes.

[EM12] presents another list of use cases qualities attributes - this time, applied to use case diagrams,. They are: consistency, correctness and completeness, fault-free, analytical, understandability. Along with the attributes, the authors had performed a systematic literature review and identified 61 unique guidelines, heuristics and rules for these format of requirements, that were synthesized and compiled into a set of 21 anti-patterns, a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.

4.1.3 Reviews and Inspections

According to the [IIB15], one way to validate those characteristics is trough a review, as they can help identify defects early in the work product life cycle, eliminating the need for expensive removal of defects discovered later in the life cycle.

[MST01] state that software review activities can be applied at many points during the software development process and can be used to discover defects in any type of deliverables or internal work products, thus having the "purification" of software artifacts as an objective. Inspection is a form of review, a rigorous defect detection process. The advantage of this process are: there is an effective mistake detection mechanism; qualitative and quantitative project feedback is given earlier; and a record of the inspection is kept, allowing the evaluation of the task performed.

[Lai02] expect that inspection results depend on inspection participants themselves and their strategies for understanding the inspected artifact. Therefore, supporting inspection participants, that is, inspectors, with particular techniques that help them detect defects

in software products, may increase the effectiveness of an inspection team most. Such techniques are referred as reading techniques. A reading technique can be defined as a series of steps or procedures whose purpose is to guide an inspector in acquiring a deep understanding of the inspected software product. The comprehension of inspected software products is a prerequisite for detecting subtle and/or complex defects, those often causing the most problems if detected in later life-cycle phases. In a sense, a reading technique can be regarded as a mechanism or strategy for the individual inspector to detect defects in the inspected product.

4.1.4 Inspection Reading Techniques

According to the [IIB15], reviews can be either formal or informal. One technique to conduct formal reviews are inspections, that includes an overview of the work product, individual review, logging the defects, team consolidation of defects, and follow-up to ensure changes were made. Ad-hoc reviews, on the other hand, is an informal technique in which the business analyst seeks informal review or assistance from a peer. Formal walkthrough (also known as team review) are another type of a formal technique that uses the individual review and team consolidation activities often seen in inspection and are used for peer reviews and for stakeholder reviews. Single issue review (also known as technical review) is another formal technique focused on either one issue or a standard in which reviewers perform a careful examination of the work product prior to a joint review session held to resolve the matter in focus. Informal walkthrough is an informal technique in which the business analyst runs through the work product in its draft state and solicits feedback and where reviewers may do minimal preparation before the joint review session. Finally, desk check is an informal technique in which a reviewer who has not been involved in the creation of the work product provides verbal or written feedback and pass around is another an informal technique in which multiple reviewers provide verbal or written feedback. The work product may be reviewed in a common copy of the work product or passed from one person to the next.

[MST01] describes many types of reviews. Formal review is accomplished directly by the customer, with the objective of providing client feedback to the developer. Internal review is the oldest method of project review, where developers distributes copies of the project to many people, chosen by the developer himself. Walkthrough is a review process that focuses on a list of problems and, through consensus, resolves them. A software inspection is the most detailed way to perform a review, as all documentation is thoroughly studied and inspected in the process many phases and necessary corrections are made in each phase in order for the process to be completed.

[Lai02] found out that ad-hoc reading and checklist-based reading are the most popular reading techniques used today for defect detection in inspection while performing a systematic review of software inspection technologies.

Ad-hoc reading offers very little reading support at all since a software product is simply given to inspectors without any direction or guidelines on how to proceed through it and what to look for. It does not, however, mean that inspection participants do not scrutinize the inspected product systematically. The word "ad-hoc" refers to the fact that no technical support is given to reviewers for the problem of how to detect defects in a software artifact and defect detection fully depends on the skill, the knowledge, and the experience of them. Training sessions may help subjects develop some of these capabilities to alleviate the lack of reading support.

Checklists offer stronger support in the form of questions, concern quality aspects of the document, inspectors are to answer while reading the document. Although reading support in the form of a list of questions is better than none, the authors debate, checklist-based reading has several weaknesses, as follows: the questions are often general and not sufficiently tailored to a particular development environment; concrete instructions on how to use a checklist are often missing, that is, it is often unclear when and based on what information an inspector is to answer a particular checklist question; questions are often limited to the detection of defects that are based on past defect information. Some principles are provided to address some of the difficulties, and other techniques were created, being PBR a scenario based reading technique one of them.

According to [MST01], a reading technique is a series of steps for the individual analysis of a software product to achieve the understanding needed for a particular task, increasing the effectiveness reviewers by providing guidelines that they can use to read a given software document and identify defects. The authors conclude that techniques attempt, to capture knowledge about best practices for defect detection into a procedure that can be followed. In their study, perspective based reading (PBR) technique, suited for the analysis of software written documentation that we want to achieve here, is described.

PBR exploits the observation that different information in the requirement is more or less important for the different uses of the document. Thus, each reviewer on a team is asked to take the perspective of a specific user of the requirements being reviewed (a tester building a test plan, a system designer building a a design, a customer evaluating if his needs are met). The technique is designed to help reviewers answer the following questions about the requirements they are inspecting:

- How do I know what information in these requirements is important to be verified ?
- Once I have found the important information, how do I find defects in that information ?

4.2 Agile Requirements

According to [HDLP15], agile software development methods take a different approach to requirements engineering and communication than the traditional requirement engineering approaches that is mostly based on formal documents and defined phases. Trough the mapping of a systematic literature reviews on the subject, the authors have pointed out some benefits and challenges for requirement engineering on agile contexts.

Due to our focus on written documentation quality, two important challenges to note are the insufficiency of the user story format and the reliance on tacit requirements knowledge. The first challenge comes from the fact that user stories do not convey enough information for software design and separate systems and subsystem are required to fill that hole. The second challenge is due to the fact that the most requirements knowledge is tacit. The next sections will describe user stories, to help on the understanding of those problems, and acceptance tests, a way to express requirements details that do not fit the user story model.

The work of [BUBE16] on how tests are used as requirements on agile contexts improves the need to better describe those tests and how they bound together with user stories. One of the companies on that research uses a behaviour-driven development approach, where requirements are documented upfront as automated acceptance test cases as part of the elicitation process and are expressed in a structured format in a way that the specification can be executable. This approach will be explained in details on the next sessions as well.

4.2.1 User Stories

According to [Rin09], traditional requirements engineering activities have become too abstract and moved away from how people ordinarily learn and communicate - too far away from storytelling, something that everyone understands intuitively. By using storytelling, the process of gathering information and structuring the requirements document would be immediately improved, as the author understands that one instinctively transform abstract knowledge into a logical structure when telling a story.

Agile methodologies are sympathetic with this thought and represents requirements using user stories. This form of requirement's representation have been created along with the extreme programming (XP) methodology, where each story describes one thing that the system needs to do [JAH00]. They're written by a customer (or a customer team) to a development team, who have the responsibility to understand the story and design, build and test a software that implement it. User Stories brings together the rights

of customers (who have the right to get the most possible value out of every programming moment, by asking for small atomic bits of functionality) and developers (who have the right to know what is needed). They're a short description of the behavior of the system from the point of view of the user of the system and are backed up with conversation and perhaps with some related detailed information.

According to [Ken00], a story represents a feature customers want in the software, a story they would like to be able to tell their friends about this great system they are using. They're some specific things that would make the system easy to use, a chunk of functionality that is of value to the customer. As a unit of functionality, the team demonstrate progress by delivering tested, integrated code that implements a story. The customer must write the story in cards, and it is up to the developers to estimate it trough collaboration with the customer.

The main format of a user story, popularized on [Coh04], is shown below:

I as a (role) want (function) so that (business value)

For [Coh04], a user story describes functionality that will be valuable to either a user or purchaser of a system or software. Each story must be written in the language of the business, not in technical jargon, so that the customer team can prioritize the stories for inclusion into iterations and releases. The author provides some reasons on why user stories are used, as follows: they emphasize verbal rather than written communication; are comprehensible by both the customer and the developers and encourage; and encourage deferring detail until the team, customers and developers, have the best understanding they are going to have about what they really need.

[LDVB15] summarizes that user stories only capture the essential elements of a requirement: *who* it is for, *what* it expects from the system, and, optionally, *why* it is important. As user stories express *what* is desired and *why* it is needed by the client, they are better compared to business or stakeholders requirements from [IIB09] and [IIB15].

4.2.2 Acceptance Tests

While [JAH00] lists only two aspects of user stories (the short description of it and the conversations around it), [Coh04] composes a User Story using three aspects:

- a written description of the story used for planning and as a reminder;
- conversations about the story that serve to flesh out the details of the story;

- tests that convey and document details and that can be used to determine when a story is complete.

Those aspects have come from [Jef01] terms Card, Conversation and Confirmation. On that work, the author described the **Card** represent customer requirements rather than document them, has just enough text to identify the requirement, and to remind everyone what the story is. The **Conversation** is an exchange of thoughts, opinions, and feelings. It is largely verbal, but can be supplemented with documents. The best supplements are examples and the best examples should be executable. They're representations of the Confirmation, a way to customers tell to developers how she will confirm that they've done what is needed in the form of acceptance tests. That confirmation, provided by those examples, is what makes possible the simple approach of card and conversation. When the conversation about a card gets down to the details of the acceptance test, the customer and programmer settle the final details of what needs to be done.

According to [Coh04], acceptance testing is the process of verifying that stories were developed such that each works exactly the way the customer team, the ones who write user stories on XP methodology, expected it to work. They're specified as soon as the iteration begins and, depending on the customer team's technical expertise, can be put into an automated testing tool. Those tests helps the customer team to communicate assumptions and expectations to the developers, by expressing the details that result from the conversations between customers and developers, and also validate that a story has been developed with the functionality the team had in mind when they wrote the story. New tests should be added as long as they add value and clarification to the story. Their scope should focus on clarifying the intent of the story to the developers and not cover all the low level validations that should happen (like invalid dates range).

[JAH00] said that acceptance tests allow the customer to know when the system works, and tell the programmers what needs to be done. On XP methodology, programmers have the right to know what is needed and customers have the right to see progress in a running system, proven to work by automated test that you specify. They provide confidence that the system really does what it needs to do.

For [Ken00], acceptance tests execution will determine whether the user stories have been successfully implemented. Also, once acceptance tests are running, the story can be discarded, since they're encoded in far more detail and accuracy in the acceptance tests, so no information will be lost if the cards are destroyed.

[Gar12] said that the hardest job in software is communicating clearly about what we want the system to do and driving the development effort with acceptance tests helps with the challenge. Two core practices are described on his book, as follows: (a) before implementing each feature, team members collaborate to create concrete examples of the feature in action and (b) then the team translates these examples into automated acceptance

tests, that become a prominent part of the team's shared, precise description of "done" for each feature. Teams that follow those practices are working on a Acceptance Test-Driven Development (**ATDD**) way.

[HS12] describe ATDD as a mixture of the documentation centric RE and the communication-focused iterative agile RE, carrying its own set of benefits and challenges. The benefits stem both from the inherent strengths that lie in discussing requirements using the acceptance tests as a mediator and its abilities for automation of the same tests. On the case study described by the authors, the technique was used to help communicate requirements (externalization) in the specification phase, a process had nothing to do with automation but with uncovering, understanding and describing requirements. In the verification phase ATDD was used to automatically verify that the code adhered to the customer requirement on an acceptance level, accompanied by manual testing for certain issues. The authors judge that ATDD saves effort in the long run, despite having a higher upfront cost due to the need of even more interaction with customers than agile development does, since the tests act as a safety harness for making changes and acts as documentation of code.

As Acceptance Tests describes the expectations that a system must show in order to demonstrate that the application is acceptable by the customer [Coh04], they can fill the documentation role of functional requirements from [IIB09] and [IIB15]. Yet, due to the fact that they're often automated test and often written by customers with their own words, they can also fill the living documentation vision proposed by [Adz11].

4.2.3 Behavior Driven Development

One way to represent acceptance tests are scenarios. [AM04] describes that the activity of building requirements scenarios encourages imagination and exploration and sets the stage for discovering unconscious and undreamed of requirements and that it is important because stories are the primary (perhaps only) means of communicating needs and desires, and providing critical feedback, to developers.

[Kan03] describes scenarios as a hypothetical story used to help a person think through a complex problem or system. Scenarios also helps to learn the product, as people don't learn well by following checklists or material that is organized for them - they learn by doing tasks that require them to investigate the product for themselves. The author also brings the idea that a scenario is an instantiation of a use case—take a specific path through the model, assigning specific values to each variable. Finally, as the scenario is a story about someone trying to accomplish something with the product under test, it can help to expose failures to deliver desired benefits.

Scenarios can also represent the key examples that describe the expected functionality, a concept introduced by [Adz11]. The living documentation presented by the author

helps to validate if the system works in the same way reflected by the documentation that represents it. Example tables are also used on the mentioned book, where one represents the system's inputs and outputs using tables where each row represents a given example. Specialized tools like the framework for integrated tests shown on [Gar12] can read those tables and use them on the software under test.

Behavior-Driven Development (**BDD**) is a set of practices that uses scenarios as an ubiquitous language to describe and model a system [Sma14]. Scenarios are expressing in a format known as Gherkin, that is designed to be both easily understandable for business stakeholders and easy to automate using dedicated tools. According to the author, bringing business and technical parties together to talk about the same document helps to build the right software (the one that meets customer needs) and to build it right (without buggy code). Those scenarios are referred as structured examples, in a similar way as [Adz11] call them key examples. Many ideas are similar on both publications. As explained by [Sma14], using conversation and examples to specify how you expect a system to behave is a core part of BDD.

BDD practitioners collaborates with the user to find concrete examples to illustrate features requested to the development team. Quite often, they also automate the execution of those scenarios. Since they are written to express the details and help the customer accept a given feature, they can be used as acceptance tests. Also, the collaboration among development team and customers that is needed to write those scenarios can be mapped as the conversations about a story that [Coh04] and [Jef01] talked about.

Those scenarios are referred as structured examples, in a similar way as [Adz11] call them key examples. Many ideas are similar on both publications. As explained by [Sma14], using conversation and examples to specify how you expect a system to behave is a core part of BDD.

In Gherkin, the requirements related to a particular feature are grouped into a single text file called a feature file. A feature file contains a short description of the feature, followed by a number of scenarios, or formalized examples of how a feature works. Each scenario is made up of a number of steps, where each step starts with one of a small number of keywords. The natural order of a scenario is *Given... When... Then...*, such as:

- Given describes the preconditions for the scenario and prepares the test environment;
- When describes the action under test;
- Then describes the expected outcomes.

One example of a scenario for a train management application that should have a feature to help a commuter to find the optimal itinerary between stations on the same line found by [Sma14]:

Given Western line trains from Emu Plains leave Parramatta for Town Hall at 7:58, 8:00, 8:02, 8:11

When I want to travel from Parramatta to Town Hall at 8:00

Then I should be told to take the 8:02 train

BDD scenarios are similar to use cases scenarios as they both describe a system behavior under certain precondition (expressed on the *Given* clauses of a BDD scenario) to achieve a certain goal (expressed on the *Then* clauses of a BDD scenario). As they're a format to express acceptance tests, they can also fill the documentation role of functional requirements from [IIB09] and [IIB15]. Finally, due to the fact that they're often automated test and are written on a ubiquitous language that should be understood by everyone in the project, they can also fill the living documentation vision proposed by [Adz11].

4.2.4 Agile Requirements Quality

According to [LDVB15], the number of methods to assess and improve user story quality is limited. Existing approaches to user story quality employ highly qualitative metrics, such as the heuristics of the INVEST (Independent-Negotiable-Valuable-Estimable-Scalable-Testable) framework described by [Coh04]. Due to that fact, [LDVB15] define additional criteria to evaluate user stories on their QUS Framework, as follows: atomic, minimal, well-formed, conflict-free, conceptually sound, problem-oriented, unambiguous, complete, explicit dependencies, full sentence, independent, scalable, uniform and unique.

For acceptance tests, our intuition is that they should meet the quality attributes that all requirements from [IIB09] and [IIB15] meet, as we compare acceptance tests with functional requirements, and the ones for use cases found by [Coc00] and [PAJ+11]. However, little work has been found to support this intuition.

Generic scenarios quality evaluation seems to be based upon subjective characteristics, like the ones found by [Kan03], or on generic guidelines, like what [AM04] has highlighted. [Kan03] describes that a test based in a scenario has five characteristics, as follows: it is (a) a story that is (b) motivating, (c) credible, (d) complex, and (e) easy to evaluate. Nevertheless, empirical evaluation of existing acceptance test cases expressed as scenarios according to his characteristics were not found. [AM04] defines a guideline that should be use for use cases and scenarios alike - his view seems to indicate they fill the same role on requirements documentation.

BDD scenarios quality attributes, on the other hand, can be only evaluated based on subjective characteristics, such as those described on [Sma14]: the scenarios steps expressiveness, focused on what goal the user want to accomplish and not on implementation

details on on screen interactions; the use of preconditions on the past tense, to make it transparent that those are actions that have already occurred in order to begin that test; the reuse of information to avoid unnecessary repetition of words; and the scenarios independence. The author specify examples of good and bad scenarios in order to demonstrate those characteristics.

4.3 Conclusion

In order to summarize the content of this chapter, we judge it necessary to directly answer the informal questions made on the beginning of it.

The quality characteristics shown on [IIB09] and [IIB15] have taught us that traditional requirements writers have many guides to help them evaluate their work [Lai02] [MST01]. Also, [PAJ+11] have shown that use cases quality criteria and guidelines are also mature enough to help practitioners writing them. Finally, user stories writers have informal guidelines (such as INVEST from [Coh04]) and an ongoing study on quality characteristics (from [LDVB15]) to help on their efforts. On the other hand, the quality of acceptance tests is still open to debate, with only a few characteristics (found by [Kan03] and [Sma14]) or generic guidelines (like the one created by [AM04]) to use, all without empirical evaluation to certify their usefulness. No inspection technique addresses acceptance tests as well, on the best of our knowledge.

On agile contexts, requirements are represented mainly as user stories according to [LDVB15]. Due to that fact, it makes sense that the authors focus their quality framework on the user story card representation from [Coh04] only, letting the details that are expressed as acceptance tests out of it. However, due to the growing importance of requirements expressed as tests shown by [BUBE16], we believe this gap need to be filled.

BDD is a format to represent acceptance tests (according to [Gar12]) that was used on one of the companies interviewed by [BUBE16] study and to which only informal characteristics (found by [Sma14]) exists to evaluate them. Since BDD scenarios represent the intended behavior of a system, the quality attributes for use cases and functional requirements could be re-used - but no study was found on that area. Also, the framework from [LDVB15] could be expanded to also cover BDD scenarios - but again, no efforts on that direction have been found. Finally, having a set of attributes to evaluate a written work without a strategy to do that may not be enough to help the writer effectvelly. Thus, there's also the need to provide a way to guide the software development team members on how to proper use those attributes in a review process.

5. RESEARCH METHODOLOGY

Due to the empirical nature of our objectives shown on Chapter 3, this research aims to create or adapt an inspection review technique to guide the quality evaluation of BDD scenarios and enhance software development team's knowledge on how good are they.

Before knowing how to reach that goal and build a proper research design, we gathered information that would improve our grasp on the main concepts involved with the quality of BDD scenarios in order to help us better define our research problem and suggest hypotheses. That approach had lead us to a better understanding of requirements quality and different formats of acceptance tests. With the possession of that knowledge, summarized on Chapter 4, we are now able to know what requirements quality attributes exists, how they are measured (through some common reading techniques) and from which other formats BDD could borrow those characteristics from. This understanding also helped us create this research plan. This is what we called Phase 1 on our research design shown on Figure 5.1

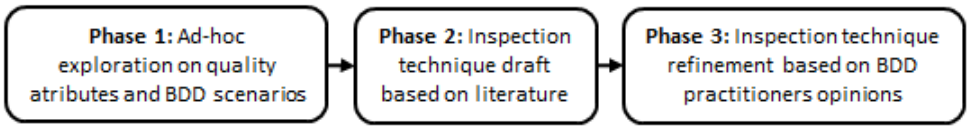

Figure 5.1 – Research Design

Since there is no standard on what is considered a "good" scenario due to the few views on this matter, this is a concept that still needs to be fully understood. A qualitative approach is suitable for situations like this, as mentioned by [Cre08]. In order to develop a technique that would help practitioners judge the quality of BDD scenarios, we first need to know what characteristics could define "good" scenarios and how they could be measured (represented on Phase 2 of our research design on Figure 5.1) and match them with the opinion of industry practitioners to clarify if they're really important when writing those scenarios - or even if other characteristics or measurements should be also worth mentioning (represented on Phase 3 of our research design on Figure 5.1). As the views of the participants will deliver us an abstract theory about how they perceive "good" scenarios, our strategy of inquiry will be based on grounded theory [Cre08]. Table 5.1 summarizes our research choices.

Table 5.1 – Research Choices

Research Design	qualitative
Strategy of Inquiry	grounded theory
Data Collection Type	interviews

6. DETAILED RESEARCH PLAN AND CURRENT PROGRESS

As described on Chapter 3, our goal is to create or adapt an inspection reading technique to assess the quality of BDD scenarios. To achieve that goal, we plan to combine the academic list of quality attributes that are useful to BDD scenarios with the practitioners opinions and our own background. The detailed research plan is summarized on Figure 6.1.

6.1 Current Progress

We've been performing a non-structured search on requirements quality, evaluation methods and known acceptance tests formats since June 2016 (represented on Figure 6.1 **steps A, B and C** respectively). The requirements formats found on that effort were use cases, user stories and bdd/gherkin and models. Since, user stories quality is already being

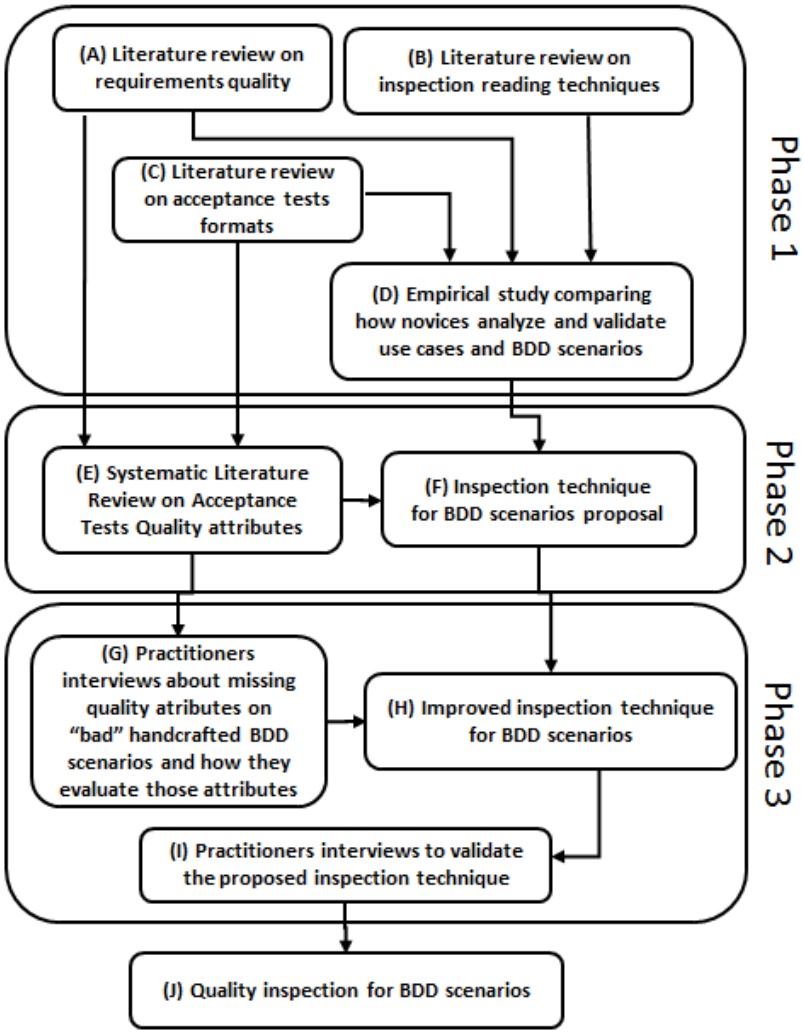

Figure 6.1 – Detailed research plan

evaluated by [LDVB15] and that our main goal was always to evaluate the quality of written BDD scenarios, we refined our goal and decided to undergo a full systematic literature review to acquire the quality attributes for acceptance tests. The rationale is that this concept can be represented by use cases, BDD scenarios or example tables as explained on Chapter 4. Therefore, the attributes found on one of them should be useful to the others as well. We hope that the characteristics found could be translated to scenarios and help us to tackle the few existing information on what attributes a "good" scenario should shown.

6.1 **step D** was planned as a moment to qualify our acquired knowledge by asking students to create use cases to a given set of software features and user stories and BDD scenarios to other set of features for a different application. After that data creation, the students were asked to validate their colleagues work using a list of quality attributes found on [IIB15] (atomic, complete, consistent, concise, feasible, unambiguous, testable, prioritized and understandable) and [Coh04] (independent, negotiable, valuable, estimable, scalable, testable). The rationale of using both lists is that the first was meant to be used with traditional requirement formats only (like use cases) while the second is more suited for user stories. With the report of this study complete, the Phase 1 will be finished, as shown on Figure 6.1.

6.2 Additional Activities and Documents

The report of the mentioned study will yield us an early version of a reading technique along with attributes, already empirically validated. However, we judge it would be wise to refine it with a systematic literature review on acceptance tests quality attributes and make sure there we used all the known state of the art knowledge. The systematic literature review is represented on 6.1 **step E** already on Phase 2 and the proposed inspection technique is shown on **step F**.

Along with this research plan, the post graduation programme of PUC university asks for a seminar to evaluate the ongoing research before the full dissertation is delivered. On that occasion, we will show the results of the systematic literature review and how it refined our early draft reading technique obtained on the **step D**. We aim to ask for at least one BDD specialist review on it before presenting it to the board on the mentioned occasion. This should conclude the Phase 2 shown on Figure 6.1.

As we mentioned on prior chapters, there is no standard on what "good" scenarios are, but we have some examples of "bad" scenarios from [Sma14]. One strategy that we devise to gather practitioners opinions is trough a series of interviews using "bad" scenarios. Using those examples, along with others taken from the empirical study and some handcrafted ones, we can ask practitioners the problems they see on them, the quality at-

tributes they have missing and how they reached those conclusions. Those interviews are represented on Figure 6.1 **step G**.

With both the quality characteristics and evaluation techniques taken from formal research and the ones taken from industry practitioners, we plan to conclude the reading technique that could assess the quality of BDD scenarios during the **step H** of Figure 6.1. Finally, it will need to be validated by industry and academic experts in order to establish its effectiveness and usefulness (**step I** of Figure 6.1).

The complete dissertation shall be delivered by the end of 2017. It will take the opinions of BDD practitioners to review and further refine the inspection reading technique and a second stage of practitioners opinions to validate it. This should conclude the Phase 3 shown on Figure 6.1 and the **step J** on the same.

6.3 Planned Timetable

Table 6.1 indicates an estimated timetable for the master's degree dissertation.

#	Activities	Months on 2017											
		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1	**Empirical study - reporting** *result mapping along with inspection technique draft*												
2	**Systematic Literature Review - planning** *search query construction and protocol review*												
3	**Systematic Literature Review - execution** *abstracts reading, paper selection, full reading*												
4	**Systematic Literature Review - reporting** *results mapping, monography writing*												
5	**Inspection technique proposal and validation with a BDD expert** *reflect monography attributed maping on inspection technique draft after validatio*												
6	**Practitioners Interview - planning** *exemples mapping and protocol review presentation*												
7	**Practitioners Interview - execution** *interviews execution, progress report presentation*												
8	**Practitioners Interview - reporting** *results mapping*												
9	**Quality inspection final changes** *reflect interviews maping on it, justify each question and changes*												
10	**Specialist Interviews to evaluate the proposed quality inspection** *justify good/bad decisions and final validations*												
11	**Dissertation final report** *results mapping, dissertation writing*												
	Phase 1 period - 2016 to Jan/2017												
	Phase 2 period - Jan to Apr												
	Phase 3 period - May to Dec												

Table 6.1 – Detailed research plan timetable

REFERENCES

[Adz11] Adzic, G. “Specification by Example: How Successful Teams Deliver the Right Software”. Greenwich, CT, USA: Manning Publications Co., 2011, 1st ed..

[AM04] Alexander, I. F.; Maiden, N. “Scenarios,Stories, Use Cases: Through the Systems Development Life-Cycle”. Wiley Publishing, 2004, 1st ed..

[BUBE16] Bjarnason, E.; Unterkalmsteiner, M.; Borg, M.; Engström, E. “A multi-case study of agile requirements engineering and the use of test cases as requirements”, *Inf. Softw. Technol.*, vol. 77–C, Sep 2016, pp. 61–79.

[Coc00] Cockburn, A. “Writing Effective Use Cases”. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000, 1st ed..

[Coh04] Cohn, M. “User Stories Applied: For Agile Software Development”. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[CR08] Cao, L.; Ramesh, B. “Agile requirements engineering practices: An empirical study”, *IEEE Software*, vol. 25–1, Jan 2008, pp. 60–67.

[Cre08] Creswell, J. W. “Research Design: Qualitative, Quantitative, and Mixed Methods Approaches”. Sage Publications Ltd., 2008, 3 ed..

[DOJ+93] Davis, A.; Overmyer, S.; Jordan, K.; Caruso, J.; Dandashi, F.; Dinh, A.; Kincaid, G.; Ledeboer, G.; Reynolds, P.; Sitaram, P.; Ta, A.; Theofanos, M. “Identifying and measuring quality in a software requirements specification”. In: Proceedings of the First International Software Metrics Symposium, 1993.

[Dun01] Duncan, R. “The quality of requirements in extreme programming”, *The Journal of Defence Software Engineering*, 2001.

[EM12] ElAttar, M.; Miller, J. “Constructing high quality use case models: a systematic review of current practices”, *Requirements Engineering*, vol. 17, 2012, pp. 187–201.

[Gar12] Gartner, M. “ATDD by Example: A Practical Guide to Acceptance Test-Driven Development”. Addison-Wesley Professional, 2012, 1st ed..

[GFL+13] Génova, G.; Fuentes, J. M.; Llorens, J.; Hurtado, O.; Moreno, V. “A framework to measure and improve the quality of textual requirements”, *Requirements Engineering*, vol. 18–1, 2013, pp. 25–41.

[HDLP15] Heikkila, V. T.; Damian, D.; Lassenius, C.; Paasivaara, M. “A mapping study on requirements engineering in agile software development”. In: Euromicro Conference on Software Engineering and Advanced Applications, 2015.

[HS12] Haugset, B.; Stalhane, T. “Automated acceptance testing as an agile requirements engineering practice”. In: Proceedings of the 45th Hawaii International Conference on System Sciences, 2012, pp. 5289–5298.

[HZ15] Heck, P.; Zaidman, A. “Quality criteria for just-in-time requirements: just enough, just-in-time?” In: JITRE, 2015.

[IIB09] IIBA. “A Guide to the Business Analysis Body of Knowledge (BABOK Guide) 2nd Edition”. International Institute of Business Analysis, 2009.

[IIB15] IIBA. “A Guide to the Business Analysis Body of Knowledge (BABOK Guide) 3rd Edition”. International Institute of Business Analysis, 2015.

[JAH00] Jeffries, R. E.; Anderson, A.; Hendrickson, C. “Extreme Programming Installed”. 2000.

[Jef01] Jeffries, R. “Essential xp: Card, conversation, confirmation”. Source: http://ronjeffries.com/xprog/articles/expcardconversationconfirmation/, Jul 2016.

[Kan03] Kaner, C. “The power of ’what if...’ and nine ways to fuel your imagination”, *STQE, the software testing and quality engineering magazine*, vol. 05, 2003.

[KB+01] Kent Beck, Alistair Cockburn, M. F.; et al.. “Agile manifesto: Manifesto for agile software development”. Visited in: 2016-08-13, Source: http://www.agilemanifesto.org/, 2001.

[Ken00] Kent Beck, M. F. “Planning Extreme Programming”. Addison-Wesley Professional, 2000, 1st ed..

[Lai02] Laitenberger, O. “A survey of software inspection technologies”, *Handbook on Software Engineering and Knowledge*, 2002.

[LDVB15] Lucassen, G.; Dalpiaz, F.; VanDerWerf, J.; Brinkkemper, S. “Forging high-quality user stories: Towards a discipline for agile requirements”, 2015, pp. 126–135.

[MST01] Melo, W.; Shull, F.; Travassos, G. H. “Software review guidelines”, *COPPE/UFRJ Systems Engineering and Computer Science Program Technical Report ES-556/01*, 2001.

[PAJ+11] Phalp, K.; Adlem, A.; Jeary, S.; Vincent, J.; Kanyaru, J. M. “The role of comprehension in requirements and implications for use case descriptions.”, *Software Quality Journal*, vol. 19–2, 2011, pp. 461–486.

[PEM03] Paetsch, F.; Eberlein, A.; Maurer, F. “Requirements engineering and agile software development”. In: Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003, pp. 308–313.

[Rin09] Rinzler, B. “Telling Stories: A Short Path to Writing Better Software Requirements”. John Wiley and Sons, 2009.

[Sma14] Smart, J. “BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle”. Shelter Island, NY: Manning Publications, 2014.