View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Aaltodoc Publication Archive

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Antti Ahonen

Unit and Integration Testing of Java:

JVM Behavior-Driven Development testing frameworks vs. JUnit

Master's Thesis
Espoo, May 18, 2017

Supervisor: Professor Casper Lassenius, Aalto University
Advisor: Mikko Pohja D.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Antti Ahonen		
Title: Unit and Integration Testing of Java: JVM Behavior-Driven Development testing frameworks vs. JUnit			
Date:	May 18, 2017	**Pages:**	vi + 118
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Casper Lassenius		
Advisor:	Mikko Pohja D.Sc. (Tech.)		

This master's thesis studied how do Behavior-Driven Development testing frameworks change the testing of Java-code compared to JUnit. The research was done as a case study. The case study was conducted in industry context at Vincit Plc, were two projects changed new unit and integration tests classes to use a new BDD-testing framework instead of JUnit. Before designing the study methods, related research and their findings were reviewed to guide the study to inspect problematic areas found in unit testing. Case study data collection methods included surveys, interviews and test code analysis.

Case study provided promising results for problematic areas highlighted by earlier research. To summarize the developer practice changes, the collected data displayed an increase in unit test case granularity. Results also displayed unanimously that BDD-testing frameworks guide to write more self-documenting tests than JUnit. The structure of BDD tests highlighted better the different parts of the test. Study also revealed that the majority of participants had easier time understanding tests and removing repetition from test code. Developer perception changes in testing included the majority of study participants enjoying writing of tests more than with JUnit. The same majority also perceived that BDD-testing frameworks promote in writing higher quality test code than JUnit. Generally new test code was perceived more understandable and maintainable than tests with JUnit, although this was not unanimous. Learning curve to be effective varied between studied frameworks. Tool support of BDD-testing frameworks for testing Java Spring Framework were found ranging from adequate to good.

In conclusion, this thesis results provide small scale evidence that BDD-testing frameworks could potentially ease the maintainability and readability of unit and integration tests while same time rising the enjoyment in testing.

Keywords:	Test automation, Unit testing, Integration testing, Behavior-Driven Development, JVM, Java, Spring Framework, JUnit, Spock, Spectrum, RSpec
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Antti Ahonen		
Työn nimi: Javan yksikkö- ja integraatiotestaus: JVM:n käyttäytymisvetoiset testaustyökalut vastaan JUnit			
Päiväys:	18. toukokuuta 2017	**Sivumäärä:**	vi + 118
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Casper Lassenius		
Ohjaaja:	Tekniikan tohtori Mikko Pohja		

Tässä diplomityössä tutkittiin, kuinka käyttäytymisvetoiset testisovelluskehykset muuttavat Java-koodin testausta verrattuna JUnit:iin. Tutkimus suoritettiin tapaustutkimuksen menetelmin Vincit Oy:ssa. Tutkimukseen valittiin kaksi projektia, joissa uudet yksikkö- ja integraatiotestausluokat kirjoitettiin käyttäytymisvetoisilla testaussovelluskehyksillä JUnit:in sijaan. Työhön liittyvät aiempien tutkimusten havainnot ohjasivat työtä tarkastelemaan näissä löydettyjä ongelmallisia alueita. Tiedonkeruukeinoina käytettiin kyselyitä, haastatteluita sekä testikoodin analyysia.

Työn tulokset osoittautuivat lupaaviksi ratkaisuksi aiemmin löydettyihin ongelmallisiin seikkoihin. Kokonaisuudessaan sovelluskehittäjien testauskäytännöissä löytyi useita muutoksia. Yksikkötestien rakenne ohjautui aiempaa hienojakoisemmaksi. Tulokset osoittivat myös yksimielisesti, että käyttäytymisvetoiset testaussovelluskehykset ohjaavat kirjoittamaan aiempaa paremmin itseänsä dokumentoivia testejä. Myös testin eri loogiset osat olivat uusien testien rakenteesta helpommin luettavissa. Suurimmalla osalla tutkimukseen osallistuneista testit olivat aiempaa helpompia ymmärtää sekä niistä oli helpompi poistaa toistoa. Suurin osa koki testien kirjoittamisen myös aiempaa nautittavampana. Valtaosa vastaajista koki uusien menetelmien ohjaavan kirjoittamaan laadukkampaa testikoodia kuin aiemmin. Yleisesti ottaen uutta testikoodia pidettiin ymmärrettävämpänä ja ylläpidettävämpänä kuin JUnit testejä, tosin ei täysin yksimielisesti. Oppismiskäyrä uusien testauskehyksien parissa vaihteli tutkittujen kehysten välillä. Java Spring-sovelluskehyksen testaustuki vaihteli riittävästä tuesta hyvään tukeen.

Kokonaisuudessaan työ tarjosi pienessä skaalassa näyttöä siitä, että käyttäytymisvetoiset testaussovelluskehykset voivat mahdollisesti helpottaa yksikkö- ja integraatiotestien ylläpidettävyyttä, luettavuutta sekä koettua nautintoa näiden parissa.

Asiasanat: testiautomaatio, yksikkötestaus, integraatiotestaus, käyttäytymisvetoinen kehitys, JVM, Java, Spring-sovelluskehys, JUnit, Spock, Spectrum, RSpec

Kieli: Englanti

Abbreviations and Acronyms

JVM	Java Virtual Machine
BDD	Behavior-Driven Development
TDD	Test-Driven Development
RE	Requirements Engineering
SQA	Software Quality Assurance
AQM	Alternative Quality Model
ATDD	Acceptance Test-Driven Development
TLD	Test Last Development
V&V	Verification & Validation
GUI	Graphical User Interface
IDE	Integrated Development Environment
DDT	Data Driven Testing
DSL	Domain Specific Language
AAA	Arrange-Act-Assert; Way to structure JUnit test methods
COTM	Count of Test Methods; Study metric for average count of test methods per tested class method
CC	Code Coverage
COA	Count of Assertions; Study metric for average count of assertions per test method
COC	Count of Comments; Study metric for average count of comments per test method
TMNWC	Test Method Name Word Count; Study metric for average count of words in test method name
DDTM	Data Driven Test Methods; Study metric for ratio of data driven test methods to standard unit and integration test methods

Contents

Chapter 1

Introduction

1.1 Motivation

Software development nowadays is many times done with an incremental and iterative agile process. Many agile methodologies aim to produce quality software that is potentially ready to use in production when the iteration is done. There rarely exists time inside the iteration for dedicated long periods of quality assurance and manual testing. Instead, the quality is build within each increment. Defect free software is one the important parts that define quality in software industry. To achieve this quality in agile context, test automation is an important tool for the developer to handle. [1]

Unit and integration testing are the testing activities that majority of agile developers work on a daily or weekly basis. Although there exists previous research studying aspects of unit testing amongst developers, they are directed towards traditional unit testing with **xUnit family** testing frameworks. These frameworks are the standard way of testing in many programming languages and they include ones such as *JUnit, NUnit* and *CppUnit* [2].

Behavior-Driven Development (BDD) was originally intended as a solution to problems Test-Driven Development (TDD) practitioners faced [3]. From there, BDD propagated to acceptance-level testing [3]. BDD in general can be described as a test firsts agile methodology aiming to provide valuable and defect free software [4]. Majority of the research regarding BDD is targeted at studying it being used as an requirements engineering (RE) tool. BDD-testing tools are also used at the implementation level in various programming languages and frameworks. There exists no hard statistics, but in many programming languages implementation level BDD testing tools like *RSpec* in Ruby and *Jasmine* in JavaScript are very popular in practicing unit and integration testing. It is reasonable to suspect that these frameworks are not always used in conjunction with the practice of

BDD but are actually used for testing purposes only.

This thesis studies these implementation level BDD-testing frameworks more closely to see, how they compare to standard *xUnit family* testing frameworks in automated unit and integration testing. Main point of interest is that if they can help with the problems that earlier research had identified in traditional unit testing frameworks. This thesis provides empirical research done in actual projects in industry context. In these projects, automated testing with *JUnit* was changed into implementation level BDD-testing framework during the project.

1.2 Problem statement

There exists research, where study participant developers declare spending approximately 40% of software development time writing new tests and debugging or fixing the code [5]. During unit testing activities, multiple previous studies have found problematic areas amongst developers. As testing, debugging and fixing form so prominent part of the developer activities, its crucial to study if these problematic areas could be alleviated. To highlight some of these problems, one of the most important one is that only around 50% of developers enjoy working with unit tests [5]. Lacking motivation in unit testing is a real problem amongst practitioners [6]. Other important findings were ones such as difficulties in writing and maintaining the tests [5]. These problematic areas are inspected in more detail later in thesis.

The context of thesis is based in Java-projects and testing Java-code. **Java Virtual Machine (JVM)** and its implementation in Java Runtime Environment produces the environment for Java-projects [7]. Main motivation for choosing JVM as the platform to study testing changes was my personal interest of wanting to learn ways to enhance Java testing. Other reasons to choose JVM as the platform for studying testing changes were *JUnit* and other programming languages available through it in addition to Java [7]. *JUnit* is a popular *xUnit family* testing framework [2], and as such, many of the problematic areas highlighted by earlier research surely affect it. The multiple programming languages through JVM allowed more options in choosing alternative implementation level BDD-testing frameworks. These frameworks could then be studied and compared against *JUnit* in different unit and integration testing aspects.

1.3 Structure of the thesis

First, chapter 2 introduces in more detail the background of JVM, the concept of software quality and practices to achieve it. Especially automated testing with agile methodologies built on top of it are examined thoroughly. Related research to thesis topic is also examined in chapter 2. Chapter 3 studies the *xUnit family* testing frameworks in detail together with various implementation level BDD-testing frameworks. Chapter 4 introduces the research questions and research hypotheses. It also explains the empirical study methods in detail. After methods, chapter 5 introduces the projects and the practical use of BDD-testing frameworks in them. Next, chapter 6 explores the study results and discussion related to earlier research in detail. Finally chapter 7 summarizes the main findings of the thesis and proposes ideas for possible future research on the thesis topic.

Chapter 2

Background

First in this section brief details of JVM are examined. Second, general concepts of software quality and software quality assurance (SQA) are explained. Third, SQA practice automated testing is inspected in more detail. After this, agile methodologies Test-Driven Development, Acceptance Test-Driven Development **(ATDD)** and Behavior-Driven Development are reviewed. For the last section in this chapter, related research on the thesis study topic is presented.

2.1 Java Virtual Machine

JVM and its programming language Java was originally designed for building software on networked devices [8]. From there, next major usage possibilities for Java came in form of Web HTML-sites. Web-sites with Java embedded programs first appeared in HotJava-browser [8]. From those days usage of Java has explored new fields and JVM itself hosts nowadays many more programming languages than just Java [7].

JVM is an **abstract computing machine** that provides instruction set and different memory areas at runtime. Current implementations of JVM have brought the environment to mobile, desktop and server devices, yet the JVM itself isn't tied to any specific technology. The basic information for JVM comes from *class* files. These files include binary JVM bytecode instructions. Instead of having Java code in class files, it compiles to these binary instructions that are run on the JVM. Figure 2.1 illustrates the runtime memory areas and the class loader system. JVM has support for *primitive* and *reference* types, from which latter enables support for referencing JVM objects. [8]

JVM is interesting target for a vast amount of programming languages, thanks to its maturity, ubiquity and performance [9]. These programming languages pass as a valid JVM language if their functionality can be expressed in a valid class file [7]. Programs written in ported dynamic languages such as *Ruby (JRuby)* or *Python (Jython)* [9] can be run on the JVM. JVM also hosts an array of new programming languages such as *Scala, Clojure* and *Groovy* [7].

The possibilities of these additional JVM programming languages are crucial part of this thesis. Later on testing Java-code is examined with different implementation level BDD-testing frameworks from various JVM languages.

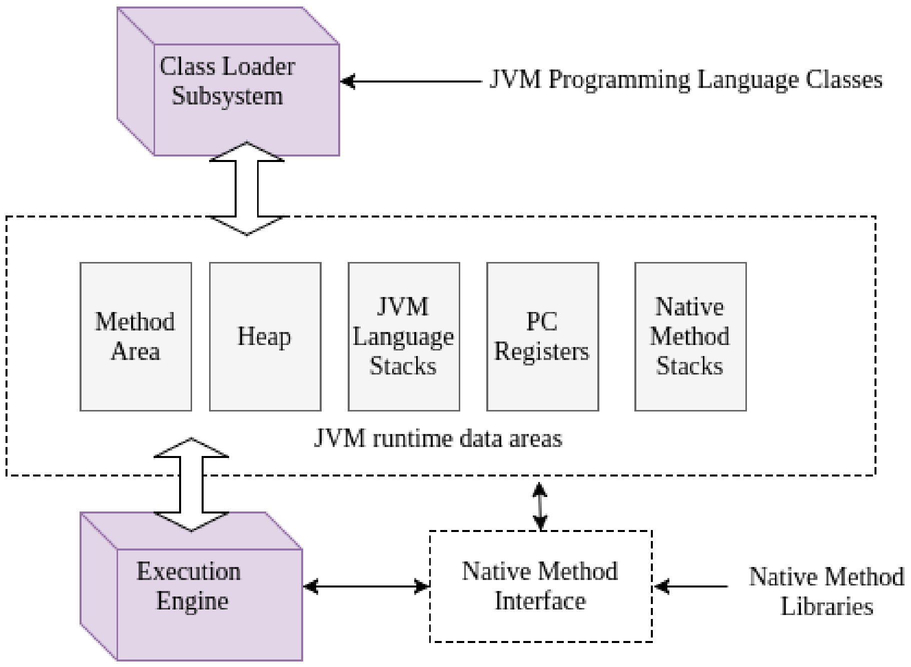


Figure 2.1: Overview of internal architecture of Java Virtual Machine

2.2 Software quality assurance

The standard definition of quality assurance states it to be: *"A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements"*. [10]

Although modern software quality assurance differs from the original standard definition, it is still definitely found at the core of SQA. To fully understand SQA and why it is important, first the software quality concept in itself is illustrated. Motivation for practicing SQA is examined second and third, the overall activities in SQA are explained briefly.

2.2.1 Definitions of quality

Quality in software development is a multifaceted entity that has had many viewpoints for a long time. Some of these earlier views include *product, transcendental, user, manufacturing* and *value based* view [11]. Because there exists so many viewpoints to what software quality is, it makes the measuring of quality hard [11]. Business goals and their priorities should determine the needed level of software quality [11] so therefore quality in itself is not set in stone, but it can alter between software services and products to fit the purpose. This can be easily demonstrated with an example of software inside a missile defence system or an online dating application. Two different systems with different goals and consequences of use, and therefore obviously different standards for needed quality.

Alternative Quality Model	
Six user levels of software quality assessment	
4	Software delights
3	Software produces no negative consequences
2	Software fits environment
1	Software fulfills all basic promises
0	Some trust, begrudging use, cynical satisfaction
-1	No trust

Table 2.1: Alternative Quality Model [12]

New alternative view on software quality by Denning [12] includes user experience directly in the core of quality. This view is called the **alternative quality model (AQM)** and it defines quality software as software that delights the end user [12]. Table 2.1 displays the full scale of AQM. While this research is mainly interested in aspects of producing traditional defect free quality software, later on in this chapter BDD is examined in detail. BDD can be seen to directly try to increase software quality on the AQM, providing value for end users and other stakeholders while at the same time minimizing defects [4].

2.2.2 Motivation for SQA

Many software projects fail but also many of them are successful. Success and failure in this context have multidimensional meaning with *technical, economic, behavioral, psychological* and *political* aspects [13]. Aggressive schedule can be usually seen as one of the primary causes for software project failure [14]. This can cause problems on many of the projects multidimensional axis, for instance in technical- and economical aspects. Projects might go over the budget, schedule, not meet the user needs and eventually be released with low quality [14]. Although many of the problems are related to requirements engineering, a lot of them are fixes or rework needed after launch [15]. If quality is measured on the AQM, then both RE and SQA are intertwined in the concept of software quality and SQA-work is essential for the success of software project. Even if software quality is only limited to mean defect free software, SQA has major role in preventing project failures.

2.2.3 Activities in SQA

Quality assurance practices and activities differ greatly in rhythm and also to a lesser degree in practices used in traditional waterfall-model software projects vs. agile projects [16]. Waterfall-projects have a rule set of their own for quality assurance, but for the topic of thesis agile methodologies and their SQA-practices are more relevant. SQA-practices can be generally categorized to *defect detection, code enhancement, verification & validation (V&V)* of artifacts, and *collaboration & communication* between stakeholders.

Defect detection can be split into two categories: **explicit** and **implicit** defect detection activities [17]. Implicit defect detection means finding defects as a secondary result when the goal was to perform another activity, such as demo presentation or giving training about the software use [17]. Explicit defect detection activities hold ones such as *testing* and artifact *inspection*, but they are found to have a lower defect detection rate than implicit ones [17]. *Continuous integration* can also be seen as a explicit defect detection as it illustrates integration defects and problems with frequent integration cycle [16].

Code enhancement activities aim to produce better design and maintainability of the codebase, and these include practices like *pair programming, refactoring* [16] and *code reviews* [18]. Although code reviews are also a form of inspection and explicit defect detection, one of their primary uses in modern code review is to share knowledge between team members [17, 18].

Verification & Validation aims to guarantee the quality of product or intermediate products. It can be used for example for design and requirement artifacts. It is a static method, which involves stakeholder(s) inspecting the artifact. It is more of a traditional waterfall software project activity, but agile practices such as code reviews can be categorized as a V&V activity. [16]

Collaboration and communication between stakeholders are used in agile methodologies frequently, being one of the cornerstones of agile practices in general [16]. Frequent customer collaboration with *on-site customer* [16] is an agile SQA-practice that establishes a foundation for many agile practices. It is essential in delivering quality on the AQM.

There are many more specific SQA practices that have their foundations on the activities mentioned in this section. Next in this thesis, testing is examined in detail through automated testing as it provides a base for agile SQA practices **TDD**, **ATDD** and **BDD**.

2.3 Automated testing

First in this section different levels of test automation are explored. Second, lower levels of test automation are explained in more detail and third, overall benefits and drawbacks of test automation are discussed.

2.3.1 Levels of automation

Design level		**Level of automation**
Requirements	$\Longrightarrow$	Acceptance testing
Functional specification	$\Longrightarrow$	System testing
Architecture design	$\Longrightarrow$	Integration testing
Component design	$\Longrightarrow$	Unit testing
$\Downarrow$		$\Uparrow$
	Code	

Table 2.2: Test last development regarding system and its design

Test automation comes at many levels of granularity regarding the system and its requirements. The four main levels for functional testing are *unit, integration, system* and *acceptance* testing. Table 2.2 illustrates how different levels of design relate to levels of test automation in traditional **Test Last Development (TLD)**. At the bottom of table 2.2 there is code, which is the result of design and foundation (in traditional automated testing) for different testing levels. Different levels of design guide the automated tests at the specified level. [19]

The introduced test automation levels are a part of functional testing. There exists also different non-functional requirements for software projects that can be automatically tested, such as *performance* and *security* testing [20]. These are not in the scope of this study, instead the functional testing levels of automated unit and integration testing are the main topic of interest.

2.3.2 Low-level testing explained

Low-level testing in the scope of this thesis means automated unit and integration level testing. They both have separate definitions and usages inside software projects, but the distinction between the two can be found confusing by many developers [21]. Table 2.2 illustrates how unit testing adheres to component design and integration testing to architecture design [19].

Unit test has many similar definitions, which all agree that it tests individual unit or collection of these units working as one [6, 22]. Unit test also has the property of being isolated from the rest of the system [22]. Unit testing can in a sense also be seen as an intersection of design, coding and debugging [23]. Dedicated practitioner of unit testing, Osherove [21], has identified important aspects of a good unit test: *trustworthiness, maintainability, readability, isolated,* has *single concern* and contains *minimal repetition.*

Isolation is an important part of unit testing. This means in practice using test doubles also known as *mocks*. Using mocks in unit tests means substituting real objects with limited functionality provided by mocks. Mocks can be configured to behave always in a specified manner. This configured behavior of mocks is called *stubbing*, in which output result of mock object can be injected from the test code. Traditional uses of mocks are for instance using them to make progress without implementing dependencies (used in TDD/BDD), isolating unit under test from dependencies or nondeterminism. [4]

Integration testing is the second, higher level of automated testing in the low-level testing scope of thesis. The definitions of integration testing state that it is a testing activity which involves multiple components [21, 22]. Osherove [21] defines integration testing as testing a unit of work with real dependencies in place: for instance database and networking. Integration tests are not usually as fasts as unit tests [21]. Many times this results from including full context of the system through dependency injection container, such as *Spring Framework* or *Guice* [24].

2.3.3 Benefits and drawbacks

Benefits	Drawbacks
Rapid feedback [1]	Can't replace manual testing [25]
Improved product quality [25, 26]	Maintaining difficulty [6, 25]
Increased test coverage [25]	Lack of skilled people [6, 25]
Increased developer confidence [25]	Hard to select correct testing strategies [25, 27]
Reduced testing time [25]	Brittle tests [27]
Shorter release cycle [27]	More development time [26]
Increased testability design [27]	Cost versus value [6]
Act as documentation [4, 23, 24]	Unmaintaned tests can lose all value [27]
Continuous regression [26]	

Table 2.3: Automated Testing Benefits & Drawbacks

Agile methodologies do not prefer or deny separate testers inside a project, but for a modern quality centered development separate testers might hinder the experience [1]. Teams without separate testers have one aspect in common, they rely heavily on test automation as the core of quality. One of the most important properties in these kind of teams is the rapid and direct feedback that test automation can provide [1]. Overall the task of test automation doesn't come with benefits only, but it has also its drawbacks.

Benefits of test automation are vast. Systematic literature review by Rafi et al. [25] has found many of them through various sources. Some of them can be highlighted: *improved product quality, test coverage increase, increase in developer confidence* and *reduced testing time.*

Automated testing was found also to allow *shorter release cycles.* This is a result of tests executed more often, and therefore allowing defects to be detected earlier with reduced cost to fix. Quality and depth of test cases also increased with automation, as less time is needed for executing them and more time is available to design the test cases. Another benefit of automated testing is the more layered, testable design of the software. To achieve the **benefits**, test automation **strategy needs to combine different approaches** and testing levels. [27]

Automated test cases can also **help to understand** the system. This is enabled by writing information how the production code should behave into the test cases and test methods. [4, 23, 24]

At unit testing level, benefits of test automation was found to include over 20% decrease in test defects [26]. Also the defects found by largely increased customer base in first 2 years of use was decreased by introducing unit testing practices [26]. Another strength of automated unit testing is also the continuous regression suite it provides [6].

Test automation isn't only beneficial compared to manual testing. In the earlier mentioned review by Rafi et al. [25], some of the **limitations** include for instance: *automation can't replace manual testing, difficulty in maintaining* and *lack of skilled people* for test automation. Usually the lack of skilled people tends to result in projects as *inapporiate test automation strategies* [25, 27]. These kind of projects usually automate testing at wrong levels, for instance automatically testing through Graphical User Interface (GUI). This can lead to brittle tests that are hard to maintain as the GUI changes [27]. Neglecting automated testing maintenance can lead to knowledge diminishing and tests that lose totally their capability to run [27]. Altogether testware maintenance is hard and this can be many times seen as tests that are not engineered with the same attention to detail as actual production code [27].

At unit testing level, drawbacks of test automation include approximately 30% more development time compared to manual testing [26]. Runeson [6] also found unit testing drawbacks similar to the afore mentioned automated testing problems by other studies. These include *unit testing of GUI*, *competency of developers* working with unit tests and *unit test maintenance* [6] . Also the cost of unit test versus the value it provides was found problematic amongst unit testing practitioners [6].

The next sections agile methodologies TDD, ATDD and BDD are all build on top of automated testing practices, operating on different levels of automation.

2.4 Test-Driven Development

2.4.1 Definition

Kent Beck [28] defines TDD as techniques aimed to produce clean code that works. This is done by driving development with automated tests: Test-Driven Development. TDD consists of two basic rules: *write new production code only when automated test fails* and *eliminate duplicated code.* These two rules are the base for **Red-Green cycle** of TDD:

1. **Red**: Design and write a small simple test. Run it and see it fail.

2. **Green**: Add code sufficient to make the earlier written test pass.

3. **Refactor**: Refactor for clean code. Improve the structure of the production code and test code. Run all tests to ensure refactoring didn't break anything.
4. **Repeat**: Repeat the cycle to add more functionality.

Design level		**Level of automation**
Component design	$\Leftrightarrow$	Unit testing
		$\Downarrow$
	Code	

Table 2.4: TDD related to system design

The tests written in TDD cycle are unit tests, providing incremental functionality to small pieces of software at a time [29]. Beck [30] also defines test-first principle to be much more than just testing. He states that TDD is also a software design technique. Table 2.4 illustrates how component design and unit testing are now in a relation where both can affect each other. Code is produced only through design captured in unit tests.

2.4.2 Benefits and drawbacks

Benefits	**Drawbacks**
Increased external quality [29, 31]	Increased development time [26, 29, 31]
Increased test coverage [29]	Requires high discipline [32]
Tests executed more [26]	Can lead to brittle tests [4, 33, 34]
More precise test cases [26]	

Table 2.5: TDD Benefits & Drawbacks

Benefits of TDD are not totally clear and exact. On earlier systematic literature review by Kollanus [31] in 2010, there was found weak evidence regarding better **external software quality** with TDD and even less evidence for better **internal software quality** with TDD. In any case, there was still results pointing to better quality, even though evidence was not uniform. External quality was measured with two methods, passing acceptance tests done by researchers and number of defects found before reported by the customer. Internal quality had multiple metrics, such as *code coverage, number of test cases, method size, cyclomatic complexity* and so on. The metrics was found at times contradicting and consensus of internal quality was left unclear.

In 2016 systematic literature review of TDD by Bissi et al. [29], the benefits of TDD was found more uniform and quite promising. 88% of studies showed significant increase in external software quality and 76% of studies identified significant increase in internal software quality. This time external quality was measured only with acceptance tests and compared to TLD approach. Internal quality was inspected with the only common metric found in the literature: code coverage. It could be argued how good of a metric code coverage is for internal quality, but at least the production code has more instructions tested with fast and repeatable automated tests when comparing TDD to TLD.

Some of the more specific benetifs of TDD was found by Williams et al. [26] to include developers creating more tests that are executed more frequently. This seems to correlate with earlier mentioned increased internal quality measured by test coverage. TDD was also shown to promote more precise and accurate test cases than TLD [26].

Drawbacks of TDD was found unanimously to include increase in development time [26, 29, 31]. For example in the study done by Williams et al. [26] the increase was ranging from 15% to 35%. As earlier mentioned, Beck [30] defined TDD to be also a design technique. This aspect could make TDD too demanding practice for junior level developers [35].

TDD **requires discipline**, that even senior level developers seem to lack when using it. This is shown for example as frequent deviation from the basic rule of starting the process with the *simplest test*. Also the *refactoring* of test code is frequently omitted by developers of all experience levels. [32]

BDD literature frequently states that TDD can have a tendency to shift viewpoint in testing to verifying system state rather than behavior of it. This can lead to brittle tests that are tightly coupled to what the object is, instead of what the object does. [4, 33, 34]

In conclusion, it could be said that TDD **is not** the **silver bullet** for software development that solves all the quality problems. The two systematic reviews mentioned provide somewhat conflicting results. Nevertheless, it could be argued that in the right context benefits of TDD outweigh the drawbacks of it.

2.5 Acceptance Test-Driven Development

ATDD is an agile practice that has many forms and names: *Specification by Example, Agile Acceptance Testing, Story Testing* and obviously *Acceptance Test-Driven Development* [36]. BDD contains also ATDD [36], but as later is discovered, it is actually a superset of it.

ATDD is an collaboration tool for stakeholders of the project [36, 37]. It means driving the development with features specified with executable acceptance level tests [36, 37]. ATDD can have many formats, such as *Gherkin, Keyword-driven testing* and *tabular formats* [36].

Instead of traditional automated testing, ATDD can be seen as a mixture of documentation centric traditional RE and communication focused agile RE [37]. The key is the communication and collaboration between stakeholders [37]. The executable automated tests are a very useful byproduct. Although the name of ATDD holds the word acceptance in it, passing tests doesn't mean that the system works perfectly [36]. Instead it is only a starting point for more quality assurance work; the system is stable enough to be manually tested further [38].

2.5.1 Benefits and drawbacks

Benefits	**Drawbacks**
Frequent collaboration [37]	Demands frequent collaboration [37]
Less ambiquity, noise and over-specification in requirements [37]	Needs active customer [37]
Faster feedback [37]	High upfront cost [37]
Increased requirement traceability [39]	Works bad with constantly changing requirements [37]
Increased trust [37]	Learning curve [37]
Increased test coverage [37]	Not for all contextes [37]
Faster start for new team members [37]	Requires high discipline [37]

Table 2.6: ATDD Benefits & Drawbacks

One of the major **benefits** of ATDD is the frequent collaboration & communication between the development team and other stakeholders. This improves the understanding of requirements [37] between all people involved. ATDD can potentially reduce the *ambiquity, noise* and *over-specification* [37] in requirements through the use of shared language. ATDD also promotes faster feedback loop [37] and traceability from requirements to code [39].

Eventually the use of ATDD seems to promote high trust between stakeholders involved, although this is an incremental process taking time. ATDD also promotes better coverage for testing and reduces overhead for new people starting to work with the project. [37]

Drawbacks of ATDD include usually the needed frequent collaboration of stakeholders in it. Customers might not have the time and effort needed for low response times of ATDD. ATDD was also found to have a high upfront cost, that should scale better overtime. This is not always the case if the requirements change all the time. [37]

ATDD has learning curve for both the development team and customers, and especially the learning curve for customers could be found too high to take the method in to use. ATDD is not for all contextes and doesn't work well for all features. ATDD as its counterpart TDD are both demanding practices that require disclipline to use properly. [37]

Next section explores Behavior-Driven Development in detail to see how it relates to TDD and ATDD and how it combines the two approaches.

2.6 Behavior-Driven Development

This section will first examine the history behind BDD. Second, BDD definition is explained. Third, the BDD process and different levels in it are examined, together with tools to support them. Finally the benefits and drawbacks of BDD are discussed.

2.6.1 History

BDD is an fairly new practice from the start of 21st century with groundwork by Dan North and Dave Astels. North [3] originally introduced BDD as a solution to problems that new practitioners of TDD faced. These included aspects such as not knowing *what to test, where to start* and *how to name tests.* The big shift was to change the language used in testing; **replacing the word test with should**. This helped to make the test method names more expressive and enabled developers to start thinking more about object behavior in testing. Later on North extended his work in changing how testing is used to accommodate also acceptance level testing to BDD with **JBehave**.

Astels [33] continued with North's ideas, resulting in creation of ***RSpec***. Astels had also found out that the used language in testing was crucial and the testing language changed from test-centric vocabulary to behavior oriented. He introduced more granular way of testing, the idea of one assert per test method to produce behavior documentation of code. The vocabulary was also changed in test condition checking **from assertions to expectations**.

2.6.2 Definition

BDD is seen as an evolution of TDD and ATDD [40]. It expands on the idea of driving system design with tests. North [3] brought in the concept of *ubiquitous language* to acceptance level BDD from **Domain Driven Design**. This ubiquitous language is used throughout the development lifecycle [40] and it should be executable [3]. The main reason for ubiquitous language is to build a common understanding between stakeholders [40] through continuous communication and collaboration. Domain Driven Design aspect of BDD is also visible through implementing the software by describing its behavior from the stakeholder perspectives in the domain context [4].

BDD is a second generation agile methodology [4], as it incorporates into existing agile practices such as *Extreme Programming, Scrum* or *Kanban* [41]. It is also used iteratively & incrementally and the behavior of system should be derived from business outcomes [40]. No clear definition of BDD exists [42], but it can be seen as describing behavior of the system at all levels of granularity [4]. In addition to these, one of the core values of BDD is *"enough is enough"*; the minimal sufficient amount of effort should be given to *planning, analysis, design* and *automation* [4]. This means doing activities just enough to incrementally provide small pieces of value, but not with the expense of quality.

2.6.3 Process

The driving factor of BDD is the analysis process through user stories and their format [43]:

> **As a** {user}
> **I want** {feature}
> **so that** {value}

The full cycle of BDD starts with *outside-in* approach. First, purpose of the project is defined and after that, business outcomes or goals for stakeholders are identified. After this feature sets are described. Individual features are analyzed for feature sets. [4]

	Business outcomes			
	⇓			
Design level	Requirements	⇔	User stories	**Level of automation**
	Functional specification	⇔	Scenarios	
	Architecture design	⇔	Integration specs	
	Component design	⇔	Unit specs	
			⇓	
	Code			

Table 2.7: BDD related to system development

Value is the **driving force** to start the progress with features. To be implemented, feature must have business value related to business outcomes [3]. From there, through different levels of system, implementation is driven with tests first to specify behavior [4]. This all correlates well with quality measured by the earlier introduced alternative quality model. BDD process aims to provide features with value in use and high internal & external software quality. This happens through *collaboration, incremental delivery* and *vast & repeatable automated test suites* [4, 41]. The outcome should be quality that delights the end user, thus performing well on the AQM [12]. Table 2.7 visualizes how business outcomes and the value derived from them drive the process from outside-in through test automation levels to code.

To fully understand the different levels of automated specification illustrated in table 2.7, the whole automation level BDD cycle [4] needs to be explained:

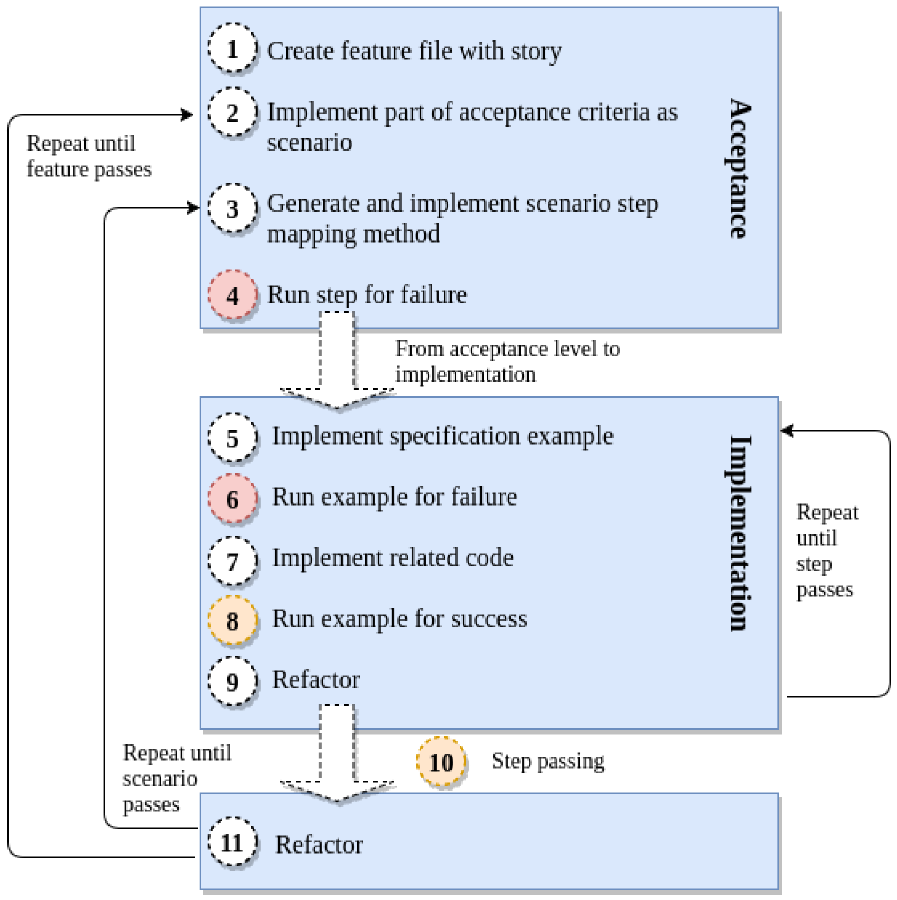

1 Create feature file with story
2 Implement part of acceptance criteria as scenario
3 Generate and implement scenario step mapping method
4 Run step for failure
Acceptance
Repeat until feature passes
From acceptance level to implementation
5 Implement specification example
6 Run example for failure
7 Implement related code
8 Run example for success
9 Refactor
Implementation
Repeat until step passes
Repeat until scenario passes
10 Step passing
11 Refactor

Figure 2.2: Automated BDD cycle

Figure 2.2 shows the BDD-cycle that can be automated with BDD tools. Steps 1 through 4 relate to acceptance level and steps from 5 to 9 are for implementation level. Step 11 is the refactoring of acceptance level code. Implementation level steps can be seen as the earlier introduced **red-green cycle of TDD**, preferably done with behavior-driven implementation level testing frameworks [41]. There exists also the outer red-green cycle, which is the outside-in aspect of BDD. The development is started with failing acceptance level, which passes later when the BDD cycle has progresses through all levels of behavior definition and its implementing code [4]. By repeating the steps, the end result will be a working feature providing value related to business outcomes [4].

2.6.4 Levels of specification

The first automation level is **acceptance level** testing [4, 40–42]. This level can be seen as a form of ATDD [36]. As earlier explained, BDD also includes the ubiquitous language [3] from Domain Driven Design and thus all ATDD does not pass as acceptance level BDD.

At acceptance level, features will be written down as feature files (exact naming depends on the used framework), where they are expressed as user stories [4]. Acceptance criteria is presented as executable scenarios [3], that usually follow predetermined ubiquitous language **Gherkin** [42]. Gherkin will be studied in more detail in the next chapter when testing framework Spock is examined. Table 2.7 shows how these stories and scenarios relate to system design. The stakeholder audience for acceptance level BDD include all stakeholders interested in development of the product [41].

Some of the BDD acceptance level tools for JVM include *Cucumber-JVM, Concordion* and *JBehave* [42]. Other environments have also BDD tools for this level, for instance *SpecFlow* for .NET and *Behave* for Python [41]. Main characteristic of acceptance level tools is business readable plain text input that is expressed with the earlier mentioned user stories and their scenarios [42].

The second automated level in BDD is **implementation level** testing [4, 40–42]. Implementation level can be seen as testing, where behavior of objects and components are described with examples [4]. This means the earlier mentioned *unit* and *integration* testing done with a new point of view. BDD is not limited to acceptancel level only, as one of the core values in it is: *"it's all behavior"* [4]. Every level can be broken down to examples describing behavior. Table 2.7 illustrates how these implementation level specifications relate to system design. Figure 2.2 shows how the two explained automation levels in BDD work together in a cycle.

Implementation level BDD can be seen to follow the principles of TDD, like the earlier mentioned red-green cycle driving the design [41]. The tests produced by BDD differ from TDD tests; they are more granular pieces of describing examples of used code [33]. They help to shift the viewpoint from test centric approach by changing the often found 1-1 mappings between test cases and test methods to classes and their methods with more descriptive naming [33].

The outcome from implementation level BDD specifications is readable, behavior oriented living documentation aimed for developers [4, 41]. Although implementation level BDD can be done with traditional xUnit tools, there exists dedicated BDD tools for writing easily **more concise** and **more expressive** low-level specifications [41]. These implementation level BDD tools provide the base for research work in this thesis, therefore they will be examined in more detail later with reviewing testing frameworks *RSpec, Spock* and *Spectrum.*

2.6.5 Benefits and drawbacks

Benefits	**Drawbacks**
Frequent collaboration* [37]	Demands frequent collaboration* [37]
Less ambiquity, noise and over-specification in requirements* [37]	Needs active customer* [37]
Faster feedback* [37]	High upfront cost* [37]
Increased requirement traceability* [39]	Works bad with constantly changing requirements* [37]
Increased trust* [37]	Learning curve* [37]
Increased test coverage* [37]	Not for all contextes* [37]
Faster start for new team members* [37]	Requires high discipline* [37]
Ubiquitous language can help in testing right aspects [42]	
Can reduce futile feature development [41]	

Table 2.8: Acceptance Level BDD Benefits & Drawbacks

* = No actual source, based on implications

Benefits (and drawbacks) of **acceptance level BDD** are almost fully the same as in ATDD. This is the result of acceptance level BDD being a form of ATDD [36]. Compared to ATDD, Acceptance level BDD introduces the ubiquitous language and it can have the effect of changing all stakeholders to think about testing to describe behavior instead of internal structures of the system [42]. The emphasis on providing value with features can also reduce production of features that are not used [41]. **Drawbacks** of acceptance level BDD are identical to the earlier mentioned ones of ATDD.

Benefits	Drawbacks
Increased external quality* [29, 31]	Increased development time* [26, 29, 31]
Increased internal quality* [29]	Requires high discipline [41]
Helps to test right aspects of component [4, 33, 34]	
More granular test cases [4, 33]	
Can help in maintenance [41]	

Table 2.9: Implementation Level BDD Benefits & Drawbacks

* = No actual source, based on implications

Benefits of **implementation level BDD** share the same characteristics of TDD. Although there exists no research on how the external and internal quality of software changes with BDD, it should share the same traits as described earlier with benefits of TDD. This is a result of implementation level BDD being an evolution of TDD [33]. As mentioned before, compared to TDD, BDD should help to focus on verifying the right aspects regarding the component. This means verifying the behavior of the component instead its structure [4, 33, 34].

Implementation level BDD aims to produce more granular and descriptive test methods and test cases than TDD [4, 33]. These test cases are also describing the behavior of production code by examples [4]. This can help the system maintenance, providing up-to-date documentation for future developers or even the original developer later on in the future [41].

Drawbacks of implementation level BDD are not clear, as empirical research on the topic is nonexistent. Because of the close relation to TDD, practicing it should introduce a growth in development time [26, 29, 31]. BDD as whole needs discipline [41], therefore it might not be a good fit for all projects and their stakeholders. It was also found that BDD is most beneficial when it is used as a holistic approach [40]. This means including both levels of specification and accommodating working practices to fully support it.

As the research on this implementation level of BDD is very limited, this thesis focuses on tools used in it. The main topic of interest is how well they could replace traditional xUnit testing family even without tests first principle. Before inspecting these JVM testing frameworks in detail, related research about low-level testing practices are reviewed.

2.7 Related research

Although BDD and BDD-testing frameworks have been around over a decade, empirical research made on the topic is limited. BDD-testing frameworks are in heavy use in certain programming languages and frameworks, such as *RSpec* in Ruby on Rails testing [44], *Jasmine* in JavaScript testing [34] and *Spock* [45] in Groovy testing. There exists no exact research on how popular practicing BDD is on the mentioned environments, but the reality probably is that these BDD implementation level frameworks are used largely only for testing purposes, not practicing BDD. The scope of this thesis is to study the changes in low-level testing from introducing BDD implementation level testing frameworks without the practice of BDD. Therefore, the previous findings in studies done on low-level testing are the most important ones regarding this thesis.

Daka and Fraser [5] studied practitioners of unit testing for used practices and problems in unit testing with a survey. They found out that developers are mainly trying to find realistic scenarios on what to test. They made important findings of developer perception towards unit testing, such as:

- Developers finding *isolating* of unit under test hard
- *Understanding* code is bigger problem than understanding test code
- Only half of the survey respondents enjoy writing unit tests
- *Maintaining* unit tests was found harder than writing them

They state that good automated unit tests could help understanding the production code. The finding of low enjoyment of practicing unit testing in developers can be seen as troublesome; Daka and Fraser notice that there exists a need for tools that rise developer enjoyment in unit testing. They also note that there is potential for unit testing research to help developers produce better tests for easier debugging and fixing of found defects. It was also discovered that there exists a need for easier maintaining of unit tests.

Li et al. [46] studied unit testing practices with a developer survey. They specialized in studying the documentation of unit testing practices. One general major problem they found out was that around 60% of practitioners found understanding of unit tests ranging from moderately difficult to very hard. Relevant findings with unit testing documention practices were:

- Developers find updated documentation and comments in test cases useful
- Writing comments to unit tests is rarely or never done

- Developers feel that tests should be self-documenting without comments

Li et al. state that for effecting maintaining of test cases, it is important for developers to understand the impact and functionality of the test case. They also observed that 89.15% percent of developers agree or strongly agree that maintaining of test cases impacts the quality of software system. They conclude that tools for supporting maintaining of unit tests could benefit unit test practitioners.

Runeson [6] studied unit testing practices in companies to define unit testing and to evaluate it. He states that companies with unclear definition for unit testing face the risk to do bad or inconsistent testing. Runeson also found out that unit testing strategies are usually emerging from developers own ambitions, instead of management policies. Related to problems in unit tests, it was found out that maintaining of unit tests takes much effort. Other major problem was developer motivation, which seemed low when working with unit tests.

Runeson also surveyed developers about documenting practices in unit testing. He found out that unit tests are documented preferably in test code rather than in text. He states that motivation to do unit testing in agile projects could include test suites functioning as specification.

Williams et al. [26] studied effectiviness of unit test automation at Microsoft. They also surveyed developers about perception towards unit testing. Around 90% of developers agree or strongly agree that unit tests are useful for regression testing. Around same percentage of developers see unit tests helpful in aiding them to produce higher quality code. 60-70% of developers find unit tests helpful in understanding other peoples code and unit tests also helped them to debug found problems.

Berner et al. [27] report observations from their own and team members experiences of automated testing in general. In their experience, test cases at all levels get corrupted easily if they are not run frequently. They become inconsistent and difficult to understand. Neglecting test case maintenance effort can result in test cases that lose their information value running capability. The cost to restore this type of test cases is very high. Berner et al. state that inappropriate testware architecture can cause the observed problems.

In master's thesis by **Laplante** [47], where she studied differences of TDD and BDD taken into use, she found out through a survey that practitioners perceive BDD specifications to produce more readable tests than TDD counterparts practiced with *xUnit family* testing frameworks. Laplante also states as interesting future work studying the use of dynamic languages, such as JRuby, for Java testing on the JVM-platform.

In conclusion, unit testing was found to have many problematic areas from developers perspective. These include aspects such as *readability, maintainability* and *enjoyment in practicing unit tests.* Many unit testing practitioners also feel the need for test code to be *self-documenting*. Implementation level BDD-testing tools are advertised to help in creating living documentation for the code and also in providing features to help maintaining the test cases [41]. Main topic of interest in later chapters of this thesis is in studying implementation level BDD-testing frameworks and if they can help with the unit test practitioner problems illustrated in this section.

Next chapter starts with first examining *xUnit family* testing tools. These tools are the traditional testing tools in use for the related research explained in this section. After that, implementation level BDD-testing frameworks are demonstrated in detail with examples to see how they differ from the xUnit family.

Chapter 3

Environment

This chapter demonstrates few of the different options available for low-level automated testing in Java-projects. First, **xUnit family testing frameworks** are explained with *JUnit* as an example of them. Second, **implementation level BDD-testing frameworks** running on top of JVM programming languages are demonstrated as an alternative for automated low-level Java-code testing.

3.1 xUnit family testing frameworks

xUnit family of testing frameworks are free, open source software for various programming languages that all share the same basic architecture. The first implementation of a xUnit test framework was *SUnit* for Smalltalk in 1999. From there the same idea was ported for Java and thus *JUnit* was born. There exists also many other *xUnit family* members, for instance *CppUnit* for C++, *NUnit* for .NET languages and *PyUnit* for Python. These unit testing frameworks are extensible with different types of extensions. Extensions can add for example integration testing capabilities for different domains. [2]

This thesis is interested in developer practices with *JUnit* and their perception towards it, therefore next the *xUnit family* architecture is examined with *JUnit*.

3.1.1 JUnit

JUnit acts as the reference implementation of the *xUnit family* and it is also the most popular instance of them [2]. It is used for Java-code testing and it can be extended for many domains in Java testing [2]. Current stable version of *JUnit* is *JUnit4* [48], but *JUnit5* is scheduled to release in Q5 of 2017 [49], providing many new features for Java testing. The examples of *JUnit* and empirical research in this thesis are all based on *JUnit4*.

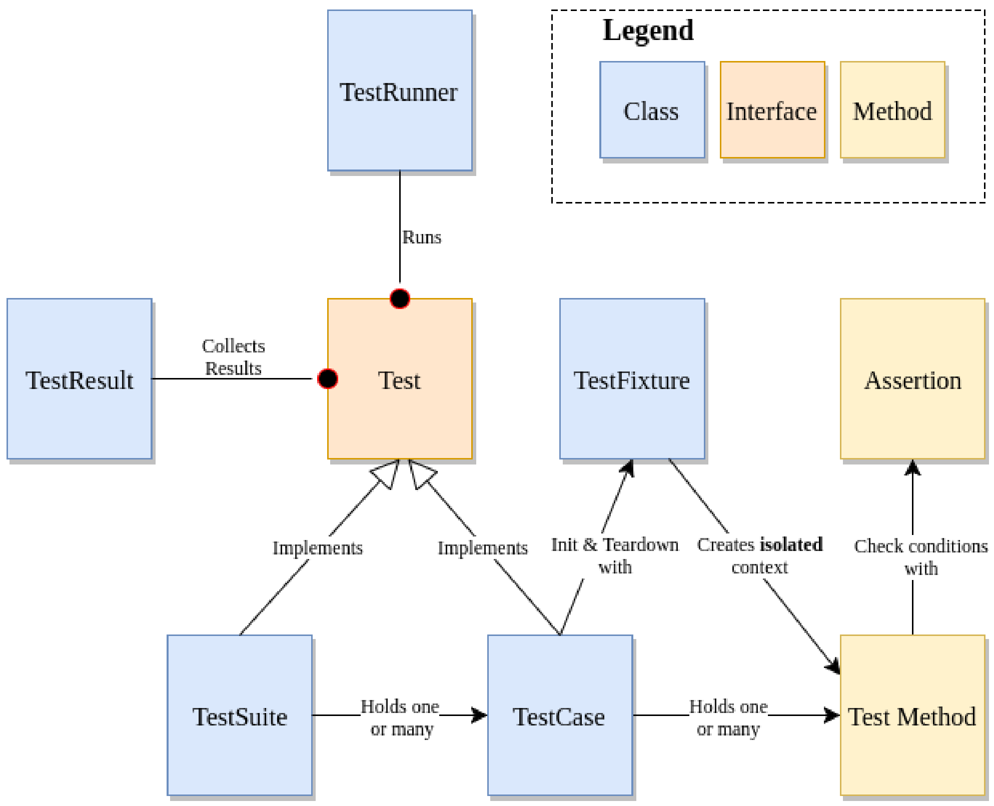


Figure 3.1: *JUnit* architecture

The basic architecture close to reference implementation of *xUnit family* [2] is explained with *JUnit3*. The architecture of *JUnit3* consists of classes *TestCase, TestRunner, TestFixture, TestSuite* and *TestResult*. **TestCase** is the unit test base class, that holds runnable **test methods**. It implements the interface **Test** and its *run()*-method. Test methods use **assertions** inside them to evaluate test conditions. **TestRunner** class is for extending *JUnit* and running multiple test cases at once and reporting them. **TestFixture** class is used to ensure test isolation and creating a separate test environment for each test method. TestFixtures provide a common shared context for test methods, where the environment is created from scratch for each test. This is enabled by providing the test case with common setup and teardown functionality for class and method level. **TestSuite** is a class made for grouping TestCases and it also implements Test. TestSuite makes it possible to run multiple TestCases and can be used with TestRunner. **TestResult** class is used to collect test method outcomes from TestSuites and TestCases. Figure 3.1 visualizes the relationships between the core classes, interfaces and methods of *JUnit* architecture. [2]

JUnit is a *"Spartan"* test framework, it contains only the mandatory features and has to use additional libraries to provide additional testing features such as as mocking or **Data Driven Testing (DDT)** [24]. In the next section, example is provided of a *JUnit4* test case extended for integration testing of *Spring Framework* domain.

3.1.2 Extending Junit for Spring Framework domain

Spring is a framework for Java, described as: *"core support for dependency injection, transaction management, web applications, data access, messaging, testing and more."* [50] It is targeted for Java enterprise applications, providing teams with a framework that allows them to primarily focus on application's business logic [50]. Spring framework ships with a lot modules and features that needs to be configured for the project [51]. **Spring Boot** is *convention-over-configuration* solution composed of the Spring Framework components, that enables rapid application development with minimal effort to get started [51].

Spring Framework integration testing can be done by extending *JUnit* with custom Spring *JUnit* runner class or by using Spring *JUnit* class and method rules. Runner class and the Spring *JUnit* rules both provide standard Spring test context for integration tests with features such as dependency injection and transactional test method execution. [52]

```
1   @RunWith(SpringJUnit4ClassRunner.class)
2   @SpringBootTest
3   public class GameServiceIntegrationTest {
4       @Autowired
5       private GameService gameService;
6
7       @Autowired
8       private GameRepository gameRepository;
```

Figure 3.2: *JUnit* extended for Spring integration testing

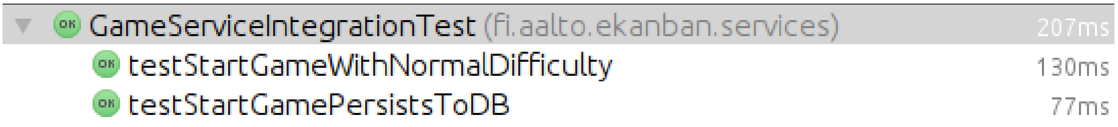
Figure 3.2 shows example of a Spring Boot *JUnit* integration test, where the context and its configuration are loaded with lines 1 and 2. Line #2 is an example of extending *JUnit* with **custom runner**. Lines 4-5 and 7-8 show examples of injected dependency via *Spring Framework*.

Figure 3.3 displays the usage of **fixture** and two **test methods**. At lines 4-8, the fixture method inits separately before each test method run two common variables used in both tests. Lines 10-11 and 19-20 display two test method definitions with the *@Test*-annotation from *JUnit4*. When used, it marks the following method as a runnable test method. Both test methods contain **assertions**. Assertions in use at lines 14-16 and 26 are *Hamcrest matchers* [53], that allows more readable descriptions for used assertions than traditional *JUnit* assertions.

```
 1  private GameDifficulty gameDifficulty;
 2  private String playerName;
 3
 4  @Before
 5  public void initGameValues() {
 6      gameDifficulty = GameDifficulty.NORMAL;
 7      playerName = "Player";
 8  }
 9
10  @Test
11  public void testStartGameWithNormalDifficulty() {
12      Game createdGame = gameService.startGame(playerName, gameDifficulty);
13
14      assertThat(createdGame, is(notNullValue()));
15      assertThat(createdGame.getPlayerName(), equalTo(playerName));
16      assertThat(createdGame.getDifficultyLevel(), equalTo(gameDifficulty));
17  }
18
19  @Test
20  public void testStartGamePersistsToDB() {
21      Integer gameCountBeforeStartGame = gameRepository.findAll().size();
22
23      gameService.startGame(playerName, gameDifficulty);
24
25      Integer gameCountAfterStartGame = gameRepository.findAll().size();
26      assertThat(gameCountAfterStartGame, is(greaterThan(gameCountBeforeStartGame)));
27  }
```

Figure 3.3: *JUnit* Spring Framework integration tests

Figure 3.4 shows the result of the test case run. Even though the example test methods start with the word test, *JUnit4* allows free format naming of the test methods that don't need to start with test. This also allows to use *JUnit* as a tool for practicing implementation level BDD with some added verbosity in test method naming [41].



```
▼ OK GameServiceIntegrationTest (fi.aalto.ekanban.services)   207ms
     OK testStartGameWithNormalDifficulty                     130ms
     OK testStartGamePersistsToDB                              77ms
```

Figure 3.4: JUnit test run results

Build configuration of *JUnit4* used in examples can be found in Appendix C figure C.1. Used build tool is Groovy-based *Gradle* [54]. *JUnit* and its extensions provide a base for automated low-level testing of Java-code. In the next section alternatives for Java-code testing are presented with implementation level BDD-testing frameworks from various JVM programming languages.

3.2 Implementation level BDD-testing frameworks for JVM

As this thesis is interested in testing Java-code, the scope of implementation level BDD is limited to testing frameworks found from JVM programming languages. As stated before, these languages hold ones such as *Ruby, Groovy, Python, Clojure* and *Scala*. Through all these languages, there exists many alternatives for both implementation and acceptance-level BDD-testing frameworks. For implementation-level, there exists two alternative approaches for practicing BDD: **xSpec family** [40] and **Gherkin family** testing frameworks.

3.2.1 xSpec family

xSpec family testing frameworks are restricted to implementation level [40]. xSpec style testing is examined in more detail in this section through ***RSpec*** via JRuby, but it also demonstrated with Java 8 -based ***Spectrum***. There exists also other possibilities for xSpec family testing in JVM enviroment, one of them being for example Scala-based *FunSpec* [55] for ScalaTest, but it is not examined in detail.

3.2.1.1 RSpec

Ruby community has been for a long time a steady advocate of TDD [44]. It is also the birth place of the first implementation level BDD-testing framework: *RSpec* [33]. Whereas first BDD framework JBehave [3] was aimed for acceptance level and all stakeholders, *RSpec* brought the behavior-driven thinking for developer audience [33].

RSpec is the founding framework in *xSpec family* of testing. In its current version 3.5, it is a mature holistic testing framework including *expectations* library (assertions), *mocking capabilities, integrations* to Ruby frameworks and the *core runner* [56]. Later on in this section, *RSpec* usage for testing Java-code both at unit and integration level for Spring framework are reviewed. Before this, the core concepts of *RSpec* are illustrated.

The main functionality of *RSpec* comes from the *rspec-core* library [56]. It includes the code **example groups** that can hold runnable specification **code examples** [4]. These examples and groups are created into a **spec** file [4]. Example groups can be initialized with keywords *describe* and *context*, which both are aliases for each other [57]. Example groups can inherit each other nested in a single file, creating nested context groups [57]. This can help on removing repetition from the test code and achieving easily readable test outputs [4]. This is enabled by **lifecycle hooks**, such as *before, after* and *around* [4]. These lifecycle hooks can be used to run separately for each example, or just once for all examples in the example group.

Specification code examples can be created with the keywords *it, specify* and *example* [57]. These examples create executable pieces of behavior of the code under specification [4]. Code examples contain **expectations**, which specify the expected behavior of the given example [4]. Expectations can be seen as assertions from *xUnit family*, but the reason for changing the language is to support better communication between stakeholders, in this case developers [4]. As BDD is an evolution of TDD, expectations was created to establish a better language for what the code to be developed *should* do, instead of verifying it with assertions [33]. The guideline for the code examples is to hold only one expectation per example [4]. This allows separate info about failing situations [4] and provides a rule **one assert per test method** for living specification [33]. This rule was originally created by Astels [33] to help TDD practitioners to change viewpoint from 1-1 relationship between test classes and methods to production classes and methods. The following table shows the relationships between *RSpec* testing terms compared to xUnit architecture components [4]:

Expectations	$\Longrightarrow$	Assertions
Code Example	$\Longrightarrow$	Test Method
Example Group	$\Longrightarrow$	Test Case
Spec File	$\Longrightarrow$	Test Suite
Lifecycle Hook	$\Longrightarrow$	Test Fixture

```
1  describe GameService do
2
3    before(:all) do
4      @gameInitService = Mockito.mock(GameInitService.java_class)
5      @playerService = Mockito.mock(PlayerService.java_class)
6      @gameOptionService = Mockito.mock(GameOptionService.java_class)
7      @gameRepository = Mockito.mock(GameRepository.java_class)
8      @game_service = GameService.new(@gameInitService, @playerService,
9                                     @gameOptionService, @gameRepository)
10   end
```

Figure 3.5: *RSpec* unit test spec file initializing

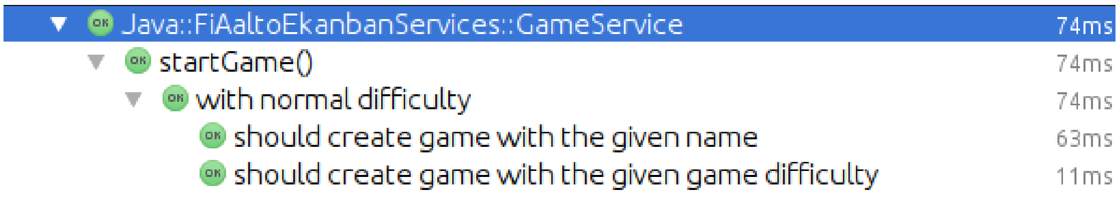
Figure 3.5 displays the creation of a spec file. The spec file example is created for unit testing of *GameService* Java-class. Line #1 shows the creation of example group with the keyword *describe*. Lines from 3 to 10 show the lifecycle hook *before*, that is used once before all code examples in the example group. In the before hook, BDD Gherkin inspired Java mocking library **Mockito** [58] is used for creating test double objects that replace the dependencies of GameService.

Figure 3.6 illustrates use of nested example groups that contain two examples in the second level nested example group. Line #1 is the first example group, that is created with keyword *describe*. The second level nested example group starts at line #9 and it is initialized with the keyword *context*. The **context** of example method is a combined from the lifecycle hooks and variable declarations of the nested example group structure.

```
1   describe 'startGame()' do
2     let(:player_name) {"Player"}
3     let(:game_difficulty) {GameDifficulty::NORMAL}
4     let(:new_game) {GameBuilder.aGame().
5                       with_player_name(player_name).
6                       with_difficulty_level(game_difficulty).
7                       build()}
8
9     context 'with normal difficulty' do
10
11      before(:each) do
12        Mockito.when(@gameInitService.getInitializedGame(
13            Mockito.any(GameDifficulty.java_class), Mockito.anyString)).thenReturn(new_game)
14        Mockito.when(@gameRepository.save(new_game)).thenReturn(new_game)
15      end
16
17      subject(:created_game) { @game_service.start_game(player_name, game_difficulty) }
18
19      it 'should create game with the given name' do
20        expect(created_game.player_name).to eq player_name
21      end
22
23      it 'should create game with the given game difficulty' do
24        expect(created_game.difficulty_level).to eq game_difficulty
25      end
26
27    end
28  end
```

Figure 3.6: *RSpec* nested example groups with code examples

RSpec lets to define the **action** of the example group with the keyword *subject* [59]. Example of used subject is at line #17. The two code examples are at lines 19-21 and 23-25. Both examples show **expectations** given with keyword *expect* at lines 20 and 24. The two expectations are separated from each other following the *one assert per test method* to create more granular documentation of behavior. As mentioned earlier, this separation also allows independently failing code examples.



```
▼ Java::FiAaltoEkanbanServices::GameService                          74ms
  ▼ startGame()                                                      74ms
    ▼ with normal difficulty                                         74ms
        should create game with the given name                       63ms
        should create game with the given game difficulty            11ms
```

Figure 3.7: *RSpec* code example run output

Figure 3.7 shows the output of code examples run in Integrated Development Environment (IDE) *Intellij Idea* [60]. The nested structure of example groups is seen with accordion elements. At the leaf of the nested tree structure is the code example and its run result. In reporting, the output of the first code example is formatted as:

Java::FiAaltoEkanbanServices::GameService startGame() with normal difficulty should create game with given name

Extending *RSpec* for *Spring Framework* domain integration testing needs to be done different route than with *JUnit*. *RSpec* runner can't use custom *JUnit* Spring runner or *JUnit* Spring context rules. Example group also can't be annotated with Spring configuration. Configuration annotation can be used with other demonstrated testing frameworks in this chapter. Figure 3.8 displays the configuration that enables Spring Framework context for *RSpec* testing. At line #6, it scans the given package for dependencies. At lines 7-9 test specific configuration is registered for the test context. Lines 17-19 display how to retrieve web server port for integration testing.

```
1   class SpringContext
2     include Singleton
3
4     def initialize
5       @ctx = AnnotationConfigEmbeddedWebApplicationContext.new
6       @ctx.scan("fi.aalto.ekanban")
7       @ctx.getEnvironment.setActiveProfiles("test")
8       @ctx.register(MongoConfiguration.java_class)
9       @ctx.register(PortConfiguration.java_class)
10      @ctx.refresh
11    end
12
13    def spring_ctx
14      @ctx
15    end
16
17    def spring_port
18      @ctx.getEmbeddedServletContainer.getPort
19    end
20  end
```

Figure 3.8: *RSpec* extension for *Spring Framework* integration testing

Figure 3.9 illustrates how-to inject dependencies for *RSpec* integration testing through *Spring Framework* dependency injection container. At line #13 the Spring Framework test context is created, and at line #14 additional dependency is retrieved through the initialized singleton integration test context object.

```
1      describe GameService do
2
3        before(:all) do
4          @ctx = SpringContext.instance.spring_ctx
5          @gameRepository = @ctx.getBean "gameRepository"
6        end
```

Figure 3.9: *Spring Framework* dependency injection example for *RSpec*

To be included in the build process, getting JRuby-based *RSpec* in use for testing Java-code needs more setup than other testing frameworks reviewed in this chapter. Appendix C figure C.4 displays the full build configuration needed to make *RSpec* a part of a *Gradle* build. To be able to run the tests in IDE *Intellij Idea*, JRuby environment with the needed *ruby gems* [61] (dependencies) installed is required.

3.2.1.2 Spectrum

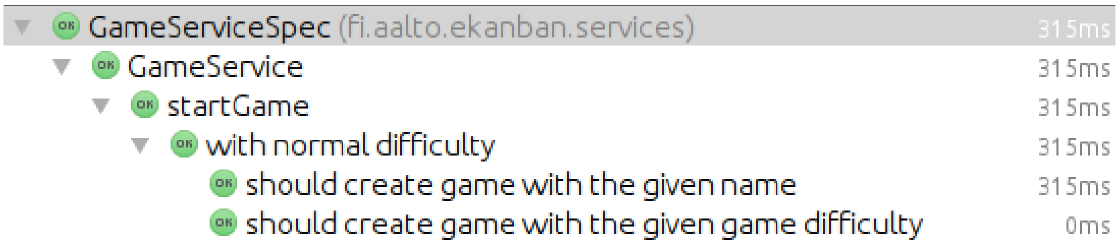
Spectrum is a Java 8 -based BDD-style **test runner for JUnit4**. It is influenced by BDD implementation level testing frameworks *Jasmine* and *RSpec*. It is used via custom runner for *JUnit*, and thus has good support for IDEs and reporting tools that are used with *JUnit*. Version 1.1.0 of *Spectrum* supports both *xSpec family* specification style and *Gherkin family* structure. This thesis inspects *Spectrum* more closely as a *RSpec* alternative for *xSpec family* style testing of Java-code. [62]

Spectrum used in examples and later on in project is version 1.0.2. It supports nested **example groups** initialized with the keyword *describe* and **code examples** with keyword *it*. Both the example groups and their code examples are build with Java 8 lambda-blocks. It has also support for **lifecycle hooks** *before* and *after*. [63]

Figure 3.10 displays the exact same Java-based specification as the JRuby *RSpec* one in figure 3.6. The main difference is the added verbosity from Java compared to Ruby. Otherwise the used example groups and their code examples match one to one. Exceptions in used terminology are missing keywords *context* and *subject*. Deviation from *RSpec* is also the usage of **assertions** with *Hamcrest matchers* instead of *expectations*. Figure 3.11 displays the output of code example runs in IDE Intellij Idea.

```
describe("startGame", () -> {

    final Supplier<GameDifficulty> gameDifficulty = let(() -> GameDifficulty.NORMAL);
    final Supplier<String> playerName = let(() -> "Player");
    final Supplier<Game> newGame = let(() -> GameBuilder.aGame()
                                            .withPlayerName(playerName.get())
                                            .withDifficultyLevel(gameDifficulty.get())
                                            .build());

    describe("with normal difficulty", () -> {

        beforeEach(() -> {
            Mockito.when(gameInitService.getInitializedGame(
                    Mockito.any(GameDifficulty.class),
                    Mockito.any(String.class))).thenReturn(newGame.get());
            Mockito.when(gameRepository.save(Mockito.any(Game.class))).thenReturn(newGame.get());
        });

        final Supplier<Game> createdGame = let(() ->
                gameService.startGame(playerName.get(), gameDifficulty.get()));

        it("should create game with the given name", () -> {
            assertThat(createdGame.get().getPlayerName(), equalTo(playerName.get()));
        });

        it("should create game with the given game difficulty", () -> {
            assertThat(createdGame.get().getDifficultyLevel(), equalTo(gameDifficulty.get()));
        });

    });
});
```

Figure 3.10: *Spectrum* nested example groups with code examples



```
GameServiceSpec (fi.aalto.ekanban.services)                        315ms
  GameService                                                      315ms
    startGame                                                      315ms
      with normal difficulty                                       315ms
        should create game with the given name                     315ms
        should create game with the given game difficulty            0ms
```

Figure 3.11: *Spectrum* code example run output

Extending *Spectrum* 1.0.2 for Spring Framework domain integration testing happens with context configuration and imperative creation of test context. At line 21 in Figure 3.12, *JUnit* is extended with custom *Spectrum*-runner. This enables the usage of example groups and code examples. Line #2 shows the annotation based configuration of the Spring Boot test context. As mentioned in section 3.1.2 about extending *JUnit* for Spring Framework, the normal extending for Spring Framework would happen with custom *JUnit* Spring runner or *JUnit* rules. Neither of these options work with the *Spectrum* version in use, but line #13 displays alternative imperative way of loading test context. Lines 5-6 and 8-9 display the dependency injection feature of *Spring Framework*.

```
 1  @RunWith(Spectrum.class)
 2  @SpringBootTest
 3  public class GameServiceIntegrationSpec {
 4
 5      @Autowired
 6      private GameService gameService;
 7
 8      @Autowired
 9      private GameRepository gameRepository;
10
11      {
12          beforeAll(() -> {
13              new TestContextManager(getClass()).prepareTestInstance(this);
14          });
```

Figure 3.12: *Spectrum* extension for Spring Framework integration testing

Build configuration for shown examples can be found in Appendix C figure C.2. The used build tool is *Gradle*.

Essentially *Spectrum* is a Java-based implementation of a **subset of features present in RSpec**. It is at early version, and is not yet mature or widely used framework. Still, it can be considered as a fully working instance of a *xSpec family* testing framework. Next section examines the alternative implementation level BDD-testing approach: *Gherkin family*.

3.2.2 Gherkin family

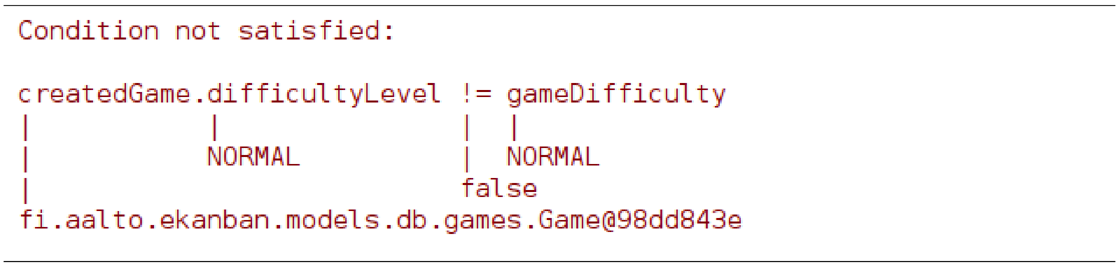
Gherkin family is a term defined in this thesis, containing BDD-testing frameworks that use predetermined ubiquitous language **Gherkin** [64]. There exists research related to usage of Gherkin in BDD-testing frameworks [42]. Gherkin family frameworks exists for both acceptance and implementation testing level [42]. For Java-code testing at implementation level with Gherkin, JVM offers for example ***Spock*** [45] via Groovy, Java 8 -based *Spectrum* [62], JRuby-based *RSpec-given* [65] for *RSpec* and Scala-based *FeatureSpec* [66] for ScalaTest. *Spock* will be used as an example for inspecting implementation level testing with a *Gherkin family* framework.

3.2.2.1 Spock

Spock is a BDD-testing framework for JVM programming language Groovy [24]. It supports testing both Java and Groovy code and it can be considered a superset of *JUnit*, as it extends the *JUnit* runner [45]. As a result of this, it is considered "enterprise ready" with *support for IDEs, JUnit rules, external tools* that use *JUnit* runner and easy *build tool integrations* [24].

Spock is capable of testing the whole automated BDD cycle from acceptance level to unit level [24]. It holds unit test support with integrated mocking and stubbing capabilities [45]. *Spock* can also be extended for integration testing of different domains [24]. *Spring Framework* domain extension is done with *spock-spring* dependency, that can be configured with the *@ContextConfiguration*-annotation of Spring [52].

Spock also supports **DDT** with tabular format readable domain specific language (DSL) [45]. DDT can drastically remove repetition from test code and makes it easier to test different parameter variations [24]. Figure 3.13 displays console debugger of *Spock* related to condition checking. This display of objects and their values and the assertion checks straight in the console can reduce the need for explicit debugging [24].



```
Condition not satisfied:

createdGame.difficultyLevel != gameDifficulty
|           |               |  |
|           NORMAL          |  NORMAL
|                           false
fi.aalto.ekanban.models.db.games.Game@98dd843e
```

Figure 3.13: *Spock* console debugger

As stated earlier, *Spock* is a BDD-testing framework with Gherkin support. Full functionality of Gherkin includes feature and scenario information described with **Given-When-Then** steps in plain text test input [64]. This makes it easier for stakeholder collaboration and is aimed at acceptance testing level. *Spock* supports full Gherkin through additional library *Pease*, but the support for it is deprecated [67]. Although *Spock* can be configured to use plain text Gherkin, its normal usage is aimed for developer audience at implementation level with test code and Gherkin descriptions mixed in together [42].

Spock terminology consists of **specification** files containing **feature methods** and Given-When-Then **blocks** inside them [45]. Figure 3.14 displays initialization

```
1   class GameServiceSpockSpec extends Specification {
2
3       @Shared GameService gameService
4       @Shared GameInitService gameInitService
5       @Shared GameRepository gameRepository
6
7       def setup() {
8           gameInitService = Mock(GameInitService)
9           gameRepository = Mock(GameRepository)
10          def playerService = Mock(PlayerService)
11          def gameOptionService = Mock(GameOptionService)
12          gameService = new GameService(gameInitService, playerService,
13                                        gameOptionService, gameRepository)
14      }
```

Figure 3.14: *Spock* specification file initialization

of a *Spock* specification file. At line #1 the specification class is created by extending the class *Spock.lang.Specification*. Lines 3-5 show use of *@Shared*-variables, which are used for long lived objects shared between feature methods [45]. Lines 7 to 14 show the use of *setup*-fixture for creating a shared, isolated context for each feature method [45]. The fixture setup is used for initializing unit testing context, where the use of *Mock*-objects can be seen. Class under test *GameService* has its dependencies replaced with test doubles.

Spock makes heavy use of Gherkin's Given-When-Then with runnable code blocks for each steps, that can contain textual description [24]:

- **Given** is a code block used to initialize the **context** of test
- **When** is a code block used to trigger the **action**, stimulus of the test
- **Then** is a code block containing assertions to verify **conditions**

JUnit testing literature also states to use this kind of structure with **Arrange-Act-Assert (AAA)**, where different parts are separated from each other with space between them [23]. As this structure is not enforced in *JUnit*, there is the considerable possibility that it will not be used [24].

Figure 3.15 illustrates the contents of a feature method and the use of Given-When-Then blocks. Block definitions are at lines 4, 10, 13, 17, 20 and 23 together with optional textual descriptions [45]. Lines 4-8 form a *Setup*-block, which is an alias for Given-block [45]. Lines 17 and 20 display the creating of *And*-block, which can be used as an alias for the previously occurred main block (Given-When-Then) for more readable structure [24].

```
 1  @Unroll
 2  def "GameService startGame() with playerName #playerName and difficulty #gameDifficulty"() {
 3
 4      setup:
 5          def newGame = GameBuilder.aGame()
 6                          .withPlayerName(playerName)
 7                          .withDifficultyLevel(gameDifficulty)
 8                          .build()
 9
10      when: "startGame() is called with playerName and gameDifficulty"
11          def createdGame = gameService.startGame(playerName, gameDifficulty)
12
13      then: "game should be initialized and persisted"
14          1 * gameInitService.getInitializedGame(gameDifficulty, playerName) >> newGame
15          1 * gameRepository.save(newGame) >> newGame
16
17      and: "game has been created with given name"
18          createdGame.playerName == playerName
19
20      and: "game has been created with given gameDifficulty"
21          createdGame.difficultyLevel != gameDifficulty
22
23      where:
24          playerName | gameDifficulty
25          "Player"   | GameDifficulty.NORMAL
26          "Pelaaja"  | GameDifficulty.NORMAL
27
28  }
```

Figure 3.15: *Spock* feature method

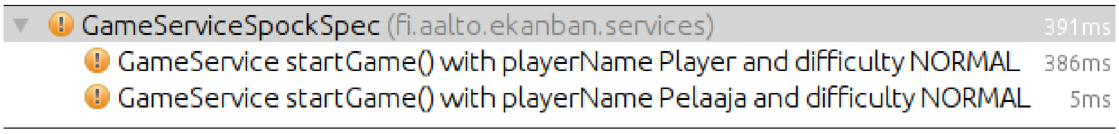
Lines 14-15 display the verifying of mock object actions and stubbing capabilities. For example, at line 14

*1 * gameInitService.getInitializedGame(gameDifficulty, playerName)*

verifies that the mock object method is called exactly one time with exact defined parameters *gameDifficulty* and *playerName*. The latter part after the verifying, *">> newGame"*, is used to stub the return value for the called mock object action.

DDT can be seen with lines 1 and 23-26. Line #1 *@Unroll* creates a separate test run for each parameter variation defined in the *Where*-block (lines 23-26). Line #24 defines the parameter names and lines 25-26 are separate situations that creates an individual feature method run. Finally the line #27 holds the definition of the feature method. The name of feature method can be inserted as a string description, allowing more easily to write information in it [24]. In the example, name holds test run output information with the embedded parameters inside it. The result of feature method run on the IDE Intellij Idea can be seen in the figure 3.16 as two separate test runs through the DDT.

To get *Spock* integrated into *Gradle* build cycle with full mocking capabilities, Spring Framework support and BDD styled HTML-reporting, it needs quite a few dependencies. The build configuration can be found in Appendix C figure C.3.



```
▼ ! GameServiceSpockSpec (fi.aalto.ekanban.services)                                      391ms
      ! GameService startGame() with playerName Player and difficulty NORMAL             386ms
      ! GameService startGame() with playerName Pelaaja and difficulty NORMAL              5ms
```

Figure 3.16: *Spock* feature method output

All the examples displayed in the figures are of *Spock* used for testing Java-code. Although at first glance the test code might look like Java, the programming language used is Groovy. Groovy is aimed to resemble Java with dynamic programming language capabilities and less verbose syntax [24]. In the given test code examples, Java production code classes are used directly from Groovy.

Next chapter explains the research around testing Java-code with different testing possibilities explained in this environment chapter. Chapter 4 will first explain what the empicical research focuses on and then how it does it.

Chapter 4

Methods

This chapter discusses the design of the study. First, research questions are introduced. Second, research hypothesis is made based on the reviewed practices & related research findings introduced in the chapter 2, together with features of studied testing frameworks explained in chapter 3. Third, the empirical study and its methodology are explained in detail.

4.1 Research questions

This thesis studies low-level testing done with *xUnit family* testing framework *JUnit* and how it changes after putting an implementation level BDD-testing framework into operation. The scope in research is testing Java-code. The following research questions are aimed to highlight the change in low-level testing after the testing framework change:

RQ1: How do BDD-testing frameworks change developer practices working with automated low-level testing compared to *JUnit* framework?

RQ2: How do BDD-testing frameworks change developer perception of working with automated low-level testing compared to *JUnit* framework?

RQ3: How do BDD-testing frameworks change written low-level test cases and test code coverage compared to *JUnit* framework?

4.2 Research hypotheses

The following hypotheses are derived from the mentioned benefits of BDD literature in chapter 2 and features of BDD testing frameworks illustrated in previous chapter. These hypotheses adhere directly to most common problems found in unit testing practitioner research mentioned in section 2.7 in chapter 2: *readibility* of unit tests is not optimal, *maintainability* of unit tests is problematic and *enjoyment* in practicing unit testing is low.

Hypothesis 1 (H1): Developers will write more granular test cases

As has been noted before in benefits of implementation level behavior-driven development, BDD-testing frameworks operating on this level aim to produce granular test cases with descriptive named test methods [4, 24, 33]. For example *RSpec* was originally intended to help the shifting of viewpoint from 1-1 relationship between test-code and only one test method per function to more granular test cases [33]. Compared to earlier average in the project, hypothesis should be visible in test cases through more measured test methods per method of component under test. In addition, the phenomena is also studied with a participant survey.

Hypothesis 2 (H2): Developers will find it easier to understand test cases

As stated earlier, implementation level BDD-testing frameworks should produce test cases of the component under test, that **describe behavior** with examples [4, 33, 34]. The close to natural language DSL used to create these behavior specifying tests with BDD-testing frameworks should have a natural tendency to "force" developers to write more descriptive tests. This is provided for example with xSpec family keywords *describe* & *it* and with Gherkin family *Given, When & Then*-steps. The context, stimulus and assertions of the tests should be more visible with these mentioned structures and the test output should be more understandable [41]. Master's thesis by Laplante had also studied that BDD specifications was perceived by developers to produce more readable tests than the ones written with xUnit testing family tools [47]. Hypothesis is measured with surveys, interview and with data from the test code analysis.

Hypothesis 3 (H3): Developers will find it easier to maintain code

Implementation level BDD specifications should produce up-to-date living documentation, which can help in overall maintenance of the the system [41]. Especially the maintaining of test cases should be easier with repetition reducing techniques provided by these BDD-testing frameworks [4, 24]. *xSpec family* nested group examples and lifecycle hooks should allow efficient repetition removal compared to traditional *xUnit family* testing frameworks. Data-Driven Testing DSL of *Spock* testing framework should allow to more easily test parameter variations and remove the need for separate test methods, therefore making it easier to maintain the test case. Hypothesis is measured with surveys and interview.

Hypothesis 4 (H4): Developers will perceive working with low-level automated testing more enjoyable.

This hypothesis is a combination of the three earlier hypotheses. By changing the used practices and language in low-level testing, the overall result should appear as more enjoyable low-level testing for developers. Especially the hypotheses 2 and 3, easier understanding and maintaining of test cases should affect how automated low-level testing is perceived. Hypothesis is measured with surveys.

4.3 Empirical study

First in this section the type of study is determined, together with rationale and objectives of it. Second, the case selection is explained with process for selecting teams and developers. Third, the data selection methods of this empirical study are explained. Finally, validity and reliability of this study are reviewed.

4.3.1 Type of study and purpose

This section explains the design of the empirical **case study** collecting evidence from multiple sources with methodological triangulation. The data was collected from multiple projects and participants with surveys and interviews. Case study data collecting was enriched with observations from multiple projects and their test code changes. Survey data can be categorized as quantative, interview data as qualitative and observation data as quantative. This study follows the pattern of deductive research, starting with background *theory* and *hypothesis* followed by *observations* and *confirmation*. This is visible through combined practices from experimental research with identified problems and predicted answer to found problems in the form of hypothesis. Case study methodology was build with suggested best practices for empirical research in software engineering [68] and case study research [69].

Rationale of this case study was to improve existing traditional low-level automated testing practices research with studying relevant technologies used in the industry. At the same time, this case study could act as a starting point for future research on the differences of traditional *xUnit family* testing frameworks and implementation level BDD-testing frameworks for larger scale studies.

Objective of the study was to compare traditional *xUnit family* testing frameworks and implementation level BDD-testing frameworks and how they change the developer practices and perception towards low-level automated testing. The change was also measured with changes in written test code. As this thesis is done at industry context, the purpose was also to determine how well these new testing frameworks in JVM context could work for future use in Java projects in the studied company.

4.3.2 Process for selecting teams and developers

The study was conducted in industry context at IT-consulting software firm **Vincit Plc**, Helsinki Mikonkatu 15 office. Related to objective of studying the automated low-level testing differences, context for the development environment was chosen as JVM. JVM provides interesting possibilities through various programming languages in use for low-level automated testing. For the selection of teams, there was limitations; team should be implementing a Java-based *Spring Framework* project with prior *JUnit* unit and integration testing in place. Limiting factor in team selection was also the amount of effort available, as my intention was to graduate in the current spring semester of 2017. This filtered out projects that could not start the study before April.

With these constraints, two projects and their teams were chosen to be studied. The teams were first introduced to implementation level BDD-testing frameworks. Both teams were presented with *RSpec, Spock* and *Spectrum*, from which they chose one technology to take into use in the projects for new created unit and integration test cases. This process is explained in more detail in chapter 5. Two months after the initial introduction and time in use for the selected BDD-testing framework, the data collecting was ended. The two months time was chosen to see, how the early adoption of new technologies had gone. The limited time available was also a practical constraint that resulted to study only for two months time. All the participants in this study were restricted to developers, as the purpose was to study developer testing practices. The projects A and B are explained in more detail in chapter 6.

4.3.3 Data collection

Selection of data included filtering of participants in the project. Participants were filtered to backend developers whom were developing with the Java *Spring Framework* and thus using *JUnit* for low-level testing purposes.

Data collection from projects is done with methodological triangulation using first degree study participant interviews, second degree participant surveys and third degree quantative test code analysis. The main tool is the second degree surveys. First, the **interview for demographic purposes** is explained. Second, the two **surveys** used to answer **RQ1 and RQ2** are examined. In addition to surveys for RQ1 and RQ2, an **interview** was conducted at the end with participants to learn more about general thoughts on the new testing frameworks used. Third, **test code analysis**, aimed to help answering **RQ3**, is defined.

4.3.3.1 Interview for demographic purposes

Participant demographics were studied with recorded semi-structured interviews [70]. The interview was constructed following the guidelines of suggested best practices [68]. The results of interviews and the analyzed demographics are shown in the chapter 6. The questions of the interview can be found from Appendix A section A.1.

4.3.3.2 JUnit survey

An online survey about *JUnit* was built to gather base information for RQ1 and RQ2, that was later on used as a reference to answer about changes in automated low-level testing practices and perception. Although the participants of the survey where currently working with *JUnit* in particular projects, they were asked to answer in more broader context of using *JUnit* through the years in different projects. The survey was designed and built using *LimeSurvey* [71]. The reason to use survey as the main tool for data collection was the related unit testing research surveys [5, 26, 46]. Some of the survey questions were interesting starting points for the built survey in this study. As there were many survey questions to begin with, it was logical to conduct the identifying of the baseline for low-level testing practices and perception with a survey. Additional motivation to use survey was to quantify results, and to provide a more specific survey question set to study problematic areas risen from related research problems.

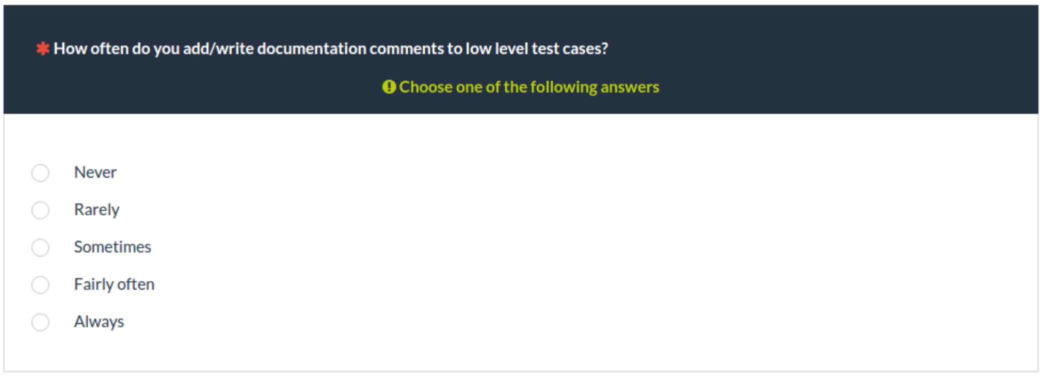
Copied survey questions were used as close to their original form as possible. The question scales were unchanged, ranging from 1 to 5 and 1 to 7 point **Likert scale** questions to one summing percentage question. The changed part was some of the words in few questions. Changed words were from *"unit"* to *"low-level"*, resulting in questions that take into consideration both unit and integration level testing.

Questions created for this research were mainly 1 to 7 point Likert scale questions. The reason to use 7 point Likert scale was to increase the discriminating power of the questions [72]. At the end of the survey **Network Promoter Score (NPS)** [73] was used to see how enthuastic and "loyal" the participants were at the beginning of the study to low-level automated testing in general and in more specific to *JUnit*. The survey questions and how they are related to research survey questions and related research can be found in Appendix B section B.1.

4.3.3.3 BDD-testing frameworks survey

Second survey was built to directly get insight on RQ1 and RQ2 about automated low-level testing and it was conducted two months after new implementation level

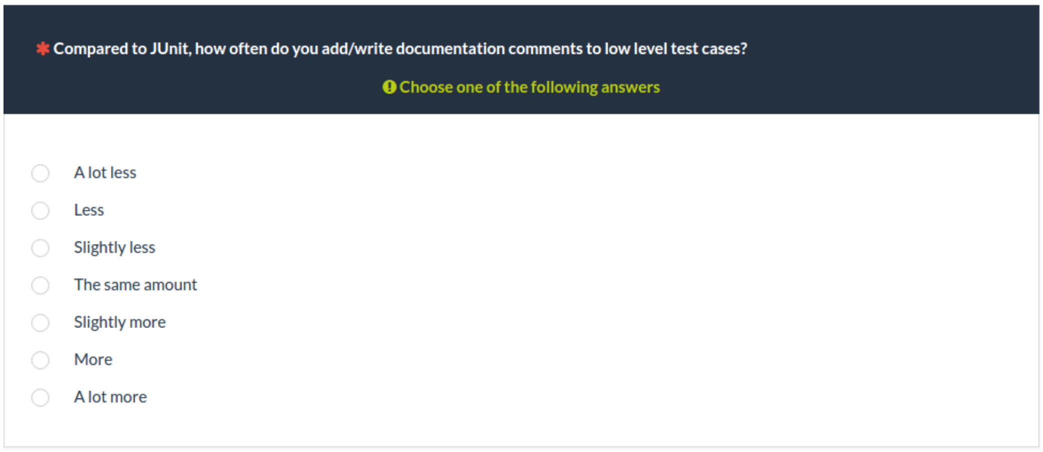
BDD-testing framework was taken into use. The second survey used all the questions (with modifications) from the first survey regarding *JUnit* testing practices and also added couple additional questions related to selected BDD-testing framework. The survey was built using *LimeSurvey*. Modifications to questions include changing the question setup to a direct comparison of *JUnit*. For example, figure 4.1 displays the original 5 point scale Likert question used in *JUnit* survey and figure 4.2 shows the comparison question used in *Spectrum* survey.


* How often do you add/write documentation comments to low level test cases?

Choose one of the following answers

- Never
- Rarely
- Sometimes
- Fairly often
- Always

Figure 4.1: JUnit survey 5 point scale Likert question


* Compared to JUnit, how often do you add/write documentation comments to low level test cases?

Choose one of the following answers

- A lot less
- Less
- Slightly less
- The same amount
- Slightly more
- More
- A lot more

Figure 4.2: Spectrum survey 7 point scale Likert comparison question

The reason to change the questions was to provide direct comparison between the test frameworks. If the same question set was repeated to gather insight on the new BDD-testing framework, there exists the risk that the participant doesn't remember what he or she answered the last time. It could result in choosing the same answer option, despite the slight change in practice or perception in testing.

The used question types were mostly 1 to 7 point Likert scale questions for the direct comparisons, combined with a few NPS questions for determining developer loyalty towards low-level automated testing in general and also towards the new BDD-testing framework. A few other type of questions were also added. The survey questions and how they are related to research survey questions and related research can be found in Appendix B section B.2.

4.3.3.4 Interview about benefits and drawbacks of new BDD-testing framework

At the end of the two month data collecting period, a loosely structured semi-structured interview was conducted with the participants to get more input on the change period. The interview holds only three basic questions to start with:

1. *What were the main benefits of the new testing framework X over JUnit?*
2. *What were the main drawbacks of the new testing framework X over JUnit?*
3. *How long was the learning curve to feel effective with framework X?*

The idea to practice these questions with a interview, instead of survey questions, was to lead the discussion to possible new topics on the subject if they rose upon interviewing. Also some clarifications were asked to interesting answers on the second survey questions. Framework X part of the question relates to chosen BDD-testing framework in the project, as it was not the same for both projects.

4.3.3.5 Test code analysis

Research question 3 is answered with test code analysis done with observation metrics from actual test data of the projects. First the baseline is calculated from existing automated low-level tests with JUnit. Second, the same values are calculated after the two months period of using an implementation level BDD-testing framework. These metric values are then compared against each other, to see how the following aspects of test code have changed:

Automated unit testing level metrics

1. *Count of test methods* ***(COTM)***: Average count of test methods per class method (COTM) is defined at unit level. This is calculated through the sum of unit test methods (UTM) divided by sum of class methods under test (CMUT). Data driven test methods add to sum of UTM as many separate methods as the the runner produces from it, for instance one data driven test method can produce 20 methods to UTM sum.

$$\sum_{i=1}^{n} \frac{UTM_i}{CMUT_i} \qquad \text{where } UTM_i \geq 1 \text{ and } CMUT_i = 1$$

2. *Code coverage* ***(CC)***: Measured code coverage for unit tests is done with *JaCoCo* [74]. The code coverage is measured with **instruction coverage** and **branch coverage** percentages [75]. Instruction coverage counts the percentage of hit-and-miss for Java byte code instructions by unit tests. Branch coverage calculates the number of executed or missed *if* and *switch* statements by unit tests.

Automated unit & integration testing level metrics

3. *Code coverage* ***(CC)***: Measured code coverage for low-level tests is done with *JaCoCo* in the same manner as described earlier in unit level code coverage definition. It includes both unit and integration level test coverage.

4. *Count of Assertions* ***(COA)***: Average count of assertions per test method (COA) is defined for low-level test methods. This is calculated with the sum of assertions (A) in low-level test methods divided by the total sum of low-level test methods (LLTM).

$$\sum_{i=1}^{n} \frac{A_i}{LLTM_i} \qquad \text{where } A_i \geq 0 \text{ and } LLTM_i = 1$$

5. *Count of Comments* ***(COC)***: Average count of comments per test method (COC) is defined for low-level test methods. This is calculated with the sum of comments (C) in low-level test methods divided by the total sum of low-level test methods (LLTM). Comments include both *inner* and *outer*

comments. Inner comments are inside the test method and outer comments preceding the test method.

$$\sum_{i=1}^{n} \frac{C_i}{LLTM_i} \qquad \text{where } C_i \geq 0 \text{ and } LLTM_i = 1$$

6. *Test method name word count* ***(TMNWC)***: Average test method name word count (TMNWC) is counted differently for different testing frameworks.

   For **JUnit** low-level test method name words are gathered through splitting the *CamelCase* [76] method name into separate words. For instance, example test method definition at line 36 in figure 3.3:

   public void testStartGameWithNormalDifficulty()

   is parsed into a 6 word test method name:

   test Start Game With Normal Difficulty

   For **xSpec family**, low-level test method name is counted starting from the first code example group description string, concatenating nested example group string descriptions and ending in the code example description string. For example from the figure 3.6, the first concatenated code example name (with word count of 11) is:

   Java::FiAaltoEkanbanServices::GameService startGame() with normal difficulty should create game with given name

   For **Spock** low-level test method name is counted from feature method string name. For example, DDT feature method name at line #27 in figure 3.15:

   GameService startGame() with playerName #playerName and difficulty #gameDifficulty

   contains 8 words.

   For each testing frameworks, TMNWC is calculated through the sum of test method name words (TMNW) divided by total sum of low-level test methods (LLTM).

$$\sum_{i=1}^{n} \frac{TMNW_i}{LLTM_i} \qquad \text{where } TMNW_i \geq 1 \text{ and } LLTM_i = 1$$

7. *Data driven test methods* ***(DDTM)***: DDTM, Ratio of data driven low-level test methods (DDLLTM) to low-level test methods (LLTM) is calculated by dividing the sum of DDLLTM by LLTM.

$$\frac{\sum_{i=0}^{n} DDLLTM_i}{\sum_{j=1}^{m} LLTM_j}$$

4.4 Validity and reliability

In this section, the different categories of validity together with reliability are analyzed with mitigated risks and remaining threats. Validity is categorized by aspects recommended by Runeson et al. [69]. Relevant threat categories in this research include *construct, internal & external validity* and *reliability* of the study. First, construct validity is analyzed. Second, internal validity is examined. Third, external validity is studied and finally reliability is discussed.

4.4.1 Construct validity

The construct validity reflects how well the studied aspects represent the original intention of the researcher [69]. Threats to construct validity include the threats to validity of the used study methods. Mitigations used for this risk in study were:

- Data collection used **methodological triangulation** with surveys, interviews and test code analysis. Quantative and qualitative data was used.
- Large parts of survey regarding the developer practices and perception towards *JUnit* were well-founded by using previous research survey questions as the starting base directly.
- Code analysis metrics such as *COTM* or *COA* were defined clearly.
- Risk in understanding of surveys was mitigated by providing support available for the participants during the answering of the survey to avoid misunderstood questions.

However, there is still a probability that the survey question was not interpreted the same way as it was intended to study the aspects of testing. Interview questions were understood seemingly correct, as the interview situations and discussions gave better insight to developer answers.

4.4.2 Internal validity

Internal validity examines the causal relations of the study [69], the presence of *bias* in the study [68]. To mitigate respondent bias, **prolonged involvement** [69] was used in studying participants of the study. During the conducting of the study, I was employed in the studied company. This enabled the access to project source code for test code analysis and also providing a trustful relationship with the study participants.

Internal validity threat includes the unexpected sources of bias [68]. The before mentioned prolonged involvement could lead to **researcher bias**. Another source of unwanted bias comes from the process of finding relevant projects and teams for the study. I first had the task of convincing participants to take a new BDD-testing framework into use in the middle of the project. This process is explained in detail in chapter 5, but it can be summerized as an enthusiastic selling of the BDD-testing frameworks and their features. This can cause some unwanted **response bias** influenced by a presence of a **"champion"** [68] driving the change and initial impressions.

4.4.3 External validity

External validity reflects how well the findings in the study can be generalized and how they could interest other people outside the study [69]. To mitigate the risks to external validity, there was kept attention in studying the BDD-testing tools in relevant industrial context in actual project settings. Study participants were also selected from senior developers with same degree experience level. **Data triangulation** was also used to study the applicability of BDD tools in different projects with multiple participant data sources.

External validity threats are related to threats that hinder the possibility to generalize the findings from this study [69]. The demographics which the study was applied can be found in section 6.1. Major threat to generalizing the findings is the **limited number of participants** involved through the projects A and B. Projects are demonstrated in detail in section 6.1, but in brief, the participant number in total in them was only 3. As the sample size is small, the results can't be generalized with certainty to all industrial developers. For example the survey regarding practices and perception towards *JUnit* can only offer insight to participant working practices, but can't be used to state a generalizations of unit and integration testing practices amongst majority of developers. The survey regarding changes in low-level testing after introduction of the implementation level BDD framework in project acts as a valuable developer experience report with its direct comparison questions, but isn't statistically relevant.

Another problem in generalization is the fact that the study was conducted within Java-projects and JVM-environment, thus this study is **specific** to the studied **environment**. Thus generalizations to other environments can't be made directly. Additional threats to external validity were the short **limited** two months period of **time** were the study was conducted and the **limited amount of new test cases** studied in the test code analysis part of the research.

4.4.4 Reliability

Reliability of this study is the aspect of the study being dependent on the researcher [69]. The study research methods and especially data collection methods are explained in detail and thus they should be able to be replicated by other researchers. Also the source code for enabling needed build configurations and test framework extensions are provided.

One aspect that hinders the reliability is the provided refactoring of *JUnit* tests to BDD-testing framework examples explained in chapter 5. This step is hard to reproduce by other research on a larger scale, as it involves time and effort not possibly available. This example providing can also cause bias in the studying by shifting the written new tests to a certain structure that supports the research hypothesis. Although providing examples of good practices should only help to produce test code closer to as originally intended by the authors of these BDD-testing framework creators.

This chapter discussed the case study details; what was studied and how it was done. Before chapter 6 and case study results, the next chapter illustrates how selected project teams chose their new implementation level BDD-testing framework to take in use.

Chapter 5

BDD frameworks in selected projects

This chapter illustrates how studied projects teams chose their implementation level BDD-testing framework to take in use for the project. The process is explained in detail with examples of *JUnit* tests refactored into example tests in different BDD-testing frameworks.

Projects are denoted as **A** and **B**. Full description of projects can be found in section 6.1 in the interview results. In brief, they both can be categorized as *Spring Framework* projects. Project A is a *Spring Boot* project, where most of the needed dependencies are bundled under *Spring Boot* configuration. Project B is a conventional *Spring Framework* project, where needed dependencies are added individually into build configuration.

Both projects and their teams were first introduced to built proof of concept examples of ***RSpec, Spock*** and ***Spectrum*** in use for an example Java *Spring Framework* project. Some of these used examples can be found in chapter 3. All my subjective pros and cons of these BDD-testing frameworks versus *JUnit* were demonstrated. *RSpec* was stripped from the potential candidates from both projects first, as it included a more complicated development and build environment configuration. Also the support in IDE's for debugging JRuby and Java code at the same time was not present. Project A developers had seemingly negative attitude towards changes in testing at first. Therefore I promised refactored examples of old *JUnit* tests to *Spectrum* and *Spock*, and tried to convince the team to switch testing framework for new tests. Project B developer had heard of *Spock* before, and thus, it was chosen as the framework to provide refactored examples in the project. First in this chapter, project A and its examples are reviewed. Second, project B and its refactored test examples are examined.

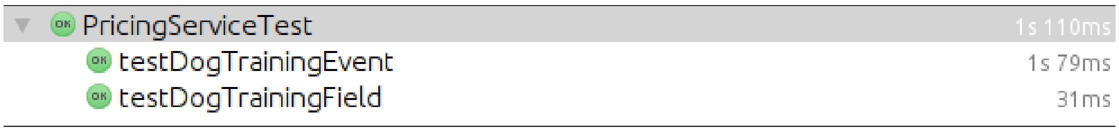
5.1 Project A

Prior the starting of study, project A had 49 test cases/files with 187 test methods of combined automated unit and integration level tests done with *JUnit*. I chose a few example test cases, which would benefit from repetition reducing techniques and better readability of the test methods. There were many more test methods and even test cases refactored, but here is provided an example of originally two *JUnit* test methods refactored into DDT feature method in *Spock* and four code examples in *Spectrum* with custom DDT technique.

```java
@Test
public void testDogTrainingEvent() {
  Application application = new Application();
  application.setType(ApplicationType.SHORT_TERM_RENTAL);
  application.setKind(ApplicationKind.DOG_TRAINING_EVENT);
  application.setStartTime(ZonedDateTime.parse("2016-11-07T06:00:00+02:00"));
  application.setEndTime(ZonedDateTime.parse("2016-12-10T05:59:59+02:00"));
  Applicant applicant = new Applicant();
  applicant.setName("Hakija");
  applicant.setType(ApplicantType.ASSOCIATION);
  application.setApplicantId(applicantDao.insert(applicant).getId());
  // association -> 50 EUR /applicationExtension
  checkPrice(application, 5000);

  applicant.setType(ApplicantType.COMPANY);
  application.setApplicantId(applicantDao.insert(applicant).getId());
  // Company -> 100 EUR /applicationExtension
  checkPrice(application, 10000);
}

@Test
public void testDogTrainingField() {
  Application application = new Application();
  application.setType(ApplicationType.SHORT_TERM_RENTAL);
  application.setKind(ApplicationKind.DOG_TRAINING_FIELD);
  application.setStartTime(ZonedDateTime.parse("2016-11-07T06:00:00+02:00"));
  application.setEndTime(ZonedDateTime.parse("2018-12-10T05:59:59+02:00"));
  Applicant applicant = new Applicant();
  applicant.setName("Hakija");
  applicant.setType(ApplicantType.ASSOCIATION);
  application.setApplicantId(applicantDao.insert(applicant).getId());
  // association -> 100 EUR /year -> 300 EUR total
  checkPrice(application, 30000);

  applicant.setType(ApplicantType.COMPANY);
  application.setApplicantId(applicantDao.insert(applicant).getId());
  // Company -> 200 EUR /year -> 600 EUR total
  checkPrice(application, 60000);
}
```

Figure 5.1: *JUnit* test methods to be refactored

Figure 5.1 displays the example *JUnit* test methods. Their test run output is displayed in the figure 5.2. When closely inspected, it was evident that the two test methods both actually included two tests inside one method, which were separated by space between them. For example test method *testDogTrainingEvent()* contains the first test within lines 3-13 and lines 15-18 are using the same *context* of the test with modifications for a new test. Both of these test methods share mostly the same test method code. Therefore, figures 5.3 and 5.5 display them refactored into tests with *Spock* and *Spectrum* that produce separate test runs with minimized repeated code.



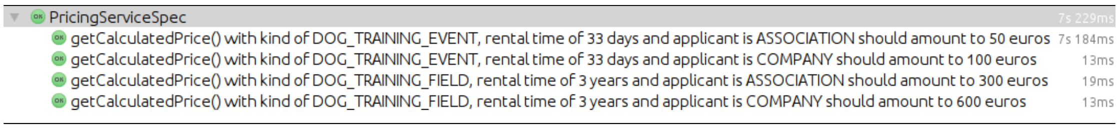
```
PricingServiceTest          1s 110ms
    testDogTrainingEvent    1s 79ms
    testDogTrainingField    31ms
```

Figure 5.2: *JUnit* test method run outputs

```
@Unroll("getCalculatedPrice() with kind of #kind, rental time of #rentTime and applicant is #applicantType should amount to #euroAmount euros")
def "Application getCalculatedPrice() with applicant"() {

    given: "application with kind #kind and rent time of #rentTime"
        def startTime = "2016-11-07T06:00:00+02:00"
        initApplicationWithGivenProperties(kind, startTime, endTime)

    and: "application has an applicant of type #applicantType"
        def applicant = new Applicant()
        applicant.setName("Hakija")
        applicant.setType(applicantType)
        application.setApplicantId(applicantDao.insert(applicant).getId())

    when: "price is updated for the given application"
        pricingService.updatePrice(application, invoiceRows)

    then: "calculated price should be #euroAmount euros"
        application.getCalculatedPrice().intValue() == intAmount

    where:
        kind                | rentTime     | euroAmount   | intAmount | applicantType |_
        DOG_TRAINING_EVENT | "33 days"    | "50"         | 5000      | ASSOCIATION   |_
        DOG_TRAINING_EVENT | "33 days"    | "100"        | 10000     | COMPANY       |_
        DOG_TRAINING_FIELD | "3 years"    | "300"        | 30000     | ASSOCIATION   |_
        DOG_TRAINING_FIELD | "3 years"    | "600"        | 60000     | COMPANY       |_

        endTime [
            "2016-12-10T05:59:59+02:00",
            "2016-12-10T05:59:59+02:00",
            "2018-12-10T05:59:59+02:00",
            "2018-12-10T05:59:59+02:00"
        ]
}
```

Figure 5.3: Figure 5.1 tests refactored into *Spock* DDT feature method

In figure 5.3, lines 20-32 together with line #1 display the data driven part of a Spock feature method. This results in a total of 4 separate test runs. Result of these runs in IDE can be seen in figure 5.4. The example shows also the use of **Given-When-Then** -blocks to structure the feature method for *context, stimulus* and *assertions* together with description comments.



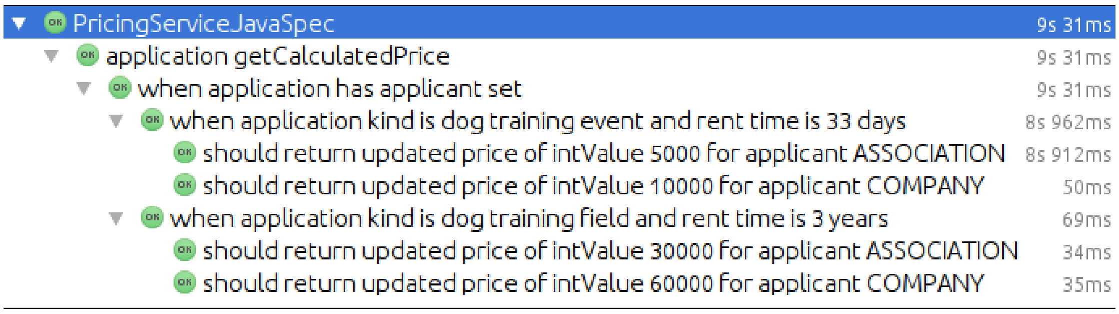
```
PricingServiceSpec                                                                                                            7s 229ms
    getCalculatedPrice() with kind of DOG_TRAINING_EVENT, rental time of 33 days and applicant is ASSOCIATION should amount to 50 euros   7s 184ms
    getCalculatedPrice() with kind of DOG_TRAINING_EVENT, rental time of 33 days and applicant is COMPANY should amount to 100 euros         13ms
    getCalculatedPrice() with kind of DOG_TRAINING_FIELD, rental time of 3 years and applicant is ASSOCIATION should amount to 300 euros     19ms
    getCalculatedPrice() with kind of DOG_TRAINING_FIELD, rental time of 3 years and applicant is COMPANY should amount to 600 euros         13ms
```

Figure 5.4: *Spock* refactored example test run output

```
describe("application getCalculatedPrice", () -> {

    beforeAll(() -> {
        startTime = "2017-06-15T08:30:00+02:00";
    });

    describe("when application has applicant set", () -> {
        describe("when application kind is dog training event and rent time is 33 days", () -> {
            beforeEach(() -> {
                endTime = "2017-07-17T08:30:00+02:00";
                initApplicationWithGivenProperties(ApplicationKind.DOG_TRAINING_EVENT, startTime,
                     endTime);
            });
            Arrays.asList(makePair(ApplicantType.ASSOCIATION, 5000),
                    makePair(ApplicantType.COMPANY,10000)).forEach(applicant -> {
                it("should return updated price of intValue "+applicant.getValue()+ " for
                     applicant "+applicant.getKey(), () -> {
                    updateApplicationPriceForGivenApplicantType(applicant.getKey());
                    updatedPrice = application.getCalculatedPrice();
                    assertEquals(updatedPrice, applicant.getValue());
                });
            });
        });
        describe("when application kind is dog training field and rent time is 3 years", () -> {
            beforeEach(() -> {
                endTime = "2019-10-15T08:30:00+02:00";
                initApplicationWithGivenProperties(ApplicationKind.DOG_TRAINING_FIELD, startTime,
                     endTime);
            });
            Arrays.asList(makePair(ApplicantType.ASSOCIATION, 30000),
                          makePair(ApplicantType.COMPANY,60000)).forEach(applicant -> {
                it("should return updated price of intValue "+applicant.getValue()+ " for
                     applicant "+applicant.getKey(), () -> {
                    updateApplicationPriceForGivenApplicantType(applicant.getKey());
                    updatedPrice = application.getCalculatedPrice();
                    assertEquals(updatedPrice, applicant.getValue());
                });
            });
        });
    });
});
```

Figure 5.5: Figure 5.1 tests refactored into *Spectrum* code examples

Figure 5.5 illustrates how the *JUnit* test methods can be refactored into Spectrum nested code **example groups** and **code examples**. The structure displayed in the figure produces 4 separate code examples. At lines 13-20 and 27-34 a custom data driven setup is made for code examples with Java 8 lambda expressions. All the description info from *describe* and *it* -blocks are concanated into test output that can be seen in figure 5.6.



```
▼ PricingServiceJavaSpec                                                              9s 31ms
  ▼ application getCalculatedPrice                                                    9s 31ms
    ▼ when application has applicant set                                              9s 31ms
      ▼ when application kind is dog training event and rent time is 33 days         8s 962ms
          should return updated price of intValue 5000 for applicant ASSOCIATION     8s 912ms
          should return updated price of intValue 10000 for applicant COMPANY           50ms
      ▼ when application kind is dog training field and rent time is 3 years            69ms
          should return updated price of intValue 30000 for applicant ASSOCIATION       34ms
          should return updated price of intValue 60000 for applicant COMPANY           35ms
```

Figure 5.6: *Spectrum* refactored examples test run output

Altogether, the given refactored examples fit flawlessly into DDT of *Spock*. *Spectrum* test code isn't as concise in the shown example, but there were some other examples were *Spectrum* produced much less repetition than *Spock* for the refactored *JUnit* test methods. Both BDD-testing frameworks produced *separate tests* for all situations, *more info on run output* and *less repetition* in test code. After the reviewing of all refactored examples and possible positive changes against *JUnit* testing, project A developers chose to take *Spectrum* in use for new unit and integration test classes. *Spectrum* being a Java library was one the main reasons why developers chose it over *Spock*.

During the observed two months time of use, *Spectrum* had couple problems that later on was found to affect parts of the research results. First, the test run output with *Maven*-build tool [77] didn't show the *code example group* and *code example* descriptions if the test failed. Only the *assertions* were showing, therefore it hindered pinpointing the problematic test at failure. Second problem was the 1.1.0 version of Spectrum, which added support for a lot of new features, but resulted in test run output structure breaking in IDE in certain situations. Therefore during the study, the version used of *Spectrum* was 1.0.2, which didn't have this problem.

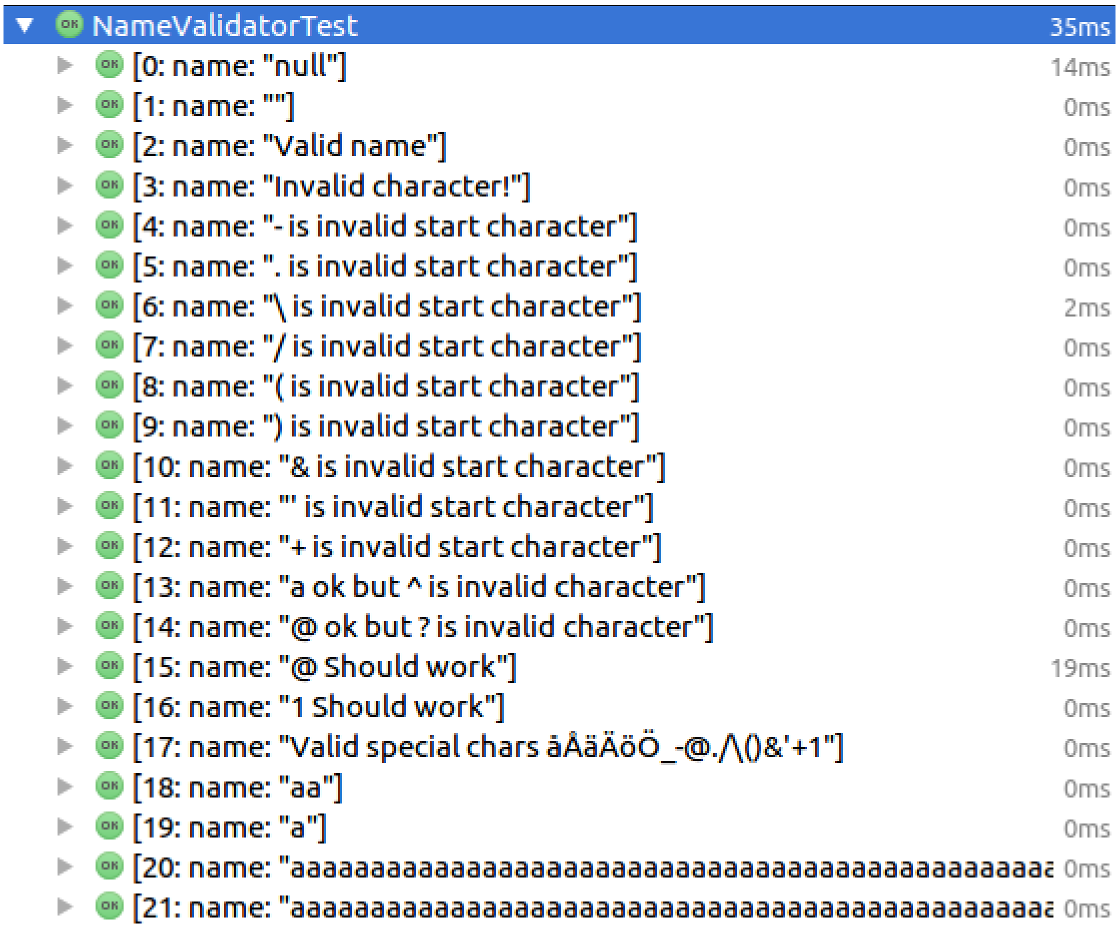
5.2 Project B

Prior the starting of study, project B had 80 test cases/files with 465 test methods of combined automated unit and integration level tests done with JUnit. Current backend developer of project B had a particular DDT *JUnit* test case in mind that he wanted to see as a refactored example for *Spock*.

```
@RunWith(Parameterized.class)
public class NameValidatorTest {
    static final int MAX_CHARACTERS = 300;

    static final String MSG_VALID = "";
    static final String MSG_INVALID_LENGTH = "invalidLength";
    static final String MSG_INVALID_FIRST_CHAR = "invalidFirstChar";
    static final String MSG_INVALID_CHAR = "invalidChar";

    private String name;
    private boolean isValid;
    private String message;

    public NameValidatorTest(String name, boolean isValid, String message) {
        this.name = name;
        this.isValid = isValid;
        this.message = message;
    }

    @Parameters(name = "{index}: name: \"{0}\"")
    public static Collection<Object[]> generateData() {
        char[] maxLength = new char[MAX_CHARACTERS];
        char[] tooLong = new char[MAX_CHARACTERS + 1];
        Arrays.fill(maxLength, 'a');
        Arrays.fill(tooLong, 'a');
        Object[][] values = new Object[][]{
                {null, false, MSG_INVALID_LENGTH},
                {"", false, MSG_INVALID_LENGTH},
                {"Valid name", true, MSG_VALID},
                {"Invalid character!", false, MSG_INVALID_CHAR},
                {"- is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {". is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {"\\ is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {"/ is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {"( is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {") is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {"& is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {"\' is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {"+ is invalid start character", false, MSG_INVALID_FIRST_CHAR},
                {"a ok but ^ is invalid character", false, MSG_INVALID_CHAR},
                {"@ ok but ? is invalid character", false, MSG_INVALID_CHAR},
                {"@ Should work", true, MSG_VALID},
                {"1 Should work", true, MSG_VALID},
                {"Valid special chars  ÃÃÃÃÃ¶Ã_ -@./\\()&\'+1", true, MSG_VALID},
                {"aa", true, MSG_VALID},
                {"a", false, MSG_INVALID_LENGTH},
                {new String(maxLength), true, MSG_VALID},
                {new String(tooLong), false, MSG_INVALID_LENGTH},
        };
        return Arrays.asList(values);
    }

    @Test
    public void testNames() throws Exception {
        NameValidator validator = new NameValidator();

        assertEquals(validator.isValid(name, null), isValid);
        assertEquals(validator.getErrorMessage(), message);
    }

}
```

Figure 5.7: *JUnit* DDT example

Figure 5.7 displays the chosen *JUnit* test case for refactoring. At line #1, *JUnit* is extended with *Parameterized* custom runner [78], which adds the DDT support for JUnit. Lines 14-18 and 20-51 display the creation of DDT test case setup in this *JUnit* example. At lines 53-59 is test method which uses this DDT setup. The result of running the test case can be seen in figure 5.8.



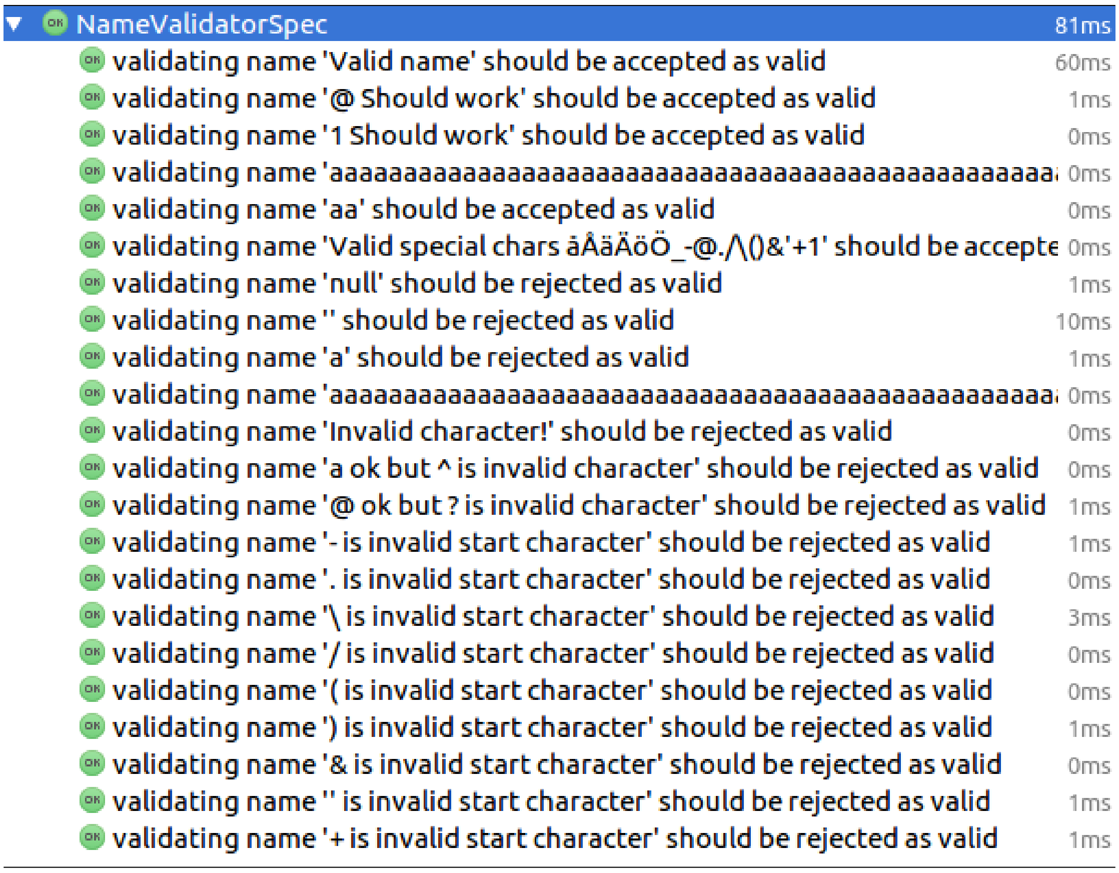
```
▼ NameValidatorTest                                              35ms
    ▶ [0: name: "null"]                                          14ms
    ▶ [1: name: ""]                                               0ms
    ▶ [2: name: "Valid name"]                                     0ms
    ▶ [3: name: "Invalid character!"]                             0ms
    ▶ [4: name: "- is invalid start character"]                   0ms
    ▶ [5: name: ". is invalid start character"]                   0ms
    ▶ [6: name: "\ is invalid start character"]                   2ms
    ▶ [7: name: "/ is invalid start character"]                   0ms
    ▶ [8: name: "( is invalid start character"]                   0ms
    ▶ [9: name: ") is invalid start character"]                   0ms
    ▶ [10: name: "& is invalid start character"]                  0ms
    ▶ [11: name: "' is invalid start character"]                  0ms
    ▶ [12: name: "+ is invalid start character"]                  0ms
    ▶ [13: name: "a ok but ^ is invalid character"]               0ms
    ▶ [14: name: "@ ok but ? is invalid character"]               0ms
    ▶ [15: name: "@ Should work"]                                19ms
    ▶ [16: name: "1 Should work"]                                 0ms
    ▶ [17: name: "Valid special chars åÅäÄöÖ_-@./\()&'+1"]        0ms
    ▶ [18: name: "aa"]                                            0ms
    ▶ [19: name: "a"]                                             0ms
    ▶ [20: name: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa     0ms
    ▶ [21: name: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa     0ms
```

Figure 5.8: *JUnit* DDT example test run output

Figure 5.9 displays the example *JUnit* test case refactored into *Spock* DDT feature method. *Spock*'s data driven DSL allows to pack the same functionality into readable concise form. At lines 31-54 the DDT table format is defined with *where*-block and at line #18 the output of individual DDT run is defined with *@Unroll*-annotation. The data driven parameters are used partly to add test output info into the feature method run. The output of feature method run can be seen in figure 5.10.

```
class NameValidatorSpec extends Specification {

   static final int MAX_CHARACTERS = 300

   static final String MSG_VALID = ""
   static final String MSG_INVALID_LENGTH = "invalidLength"
   static final String MSG_INVALID_FIRST_CHAR = "invalidFirstChar"
   static final String MSG_INVALID_CHAR = "invalidChar"

   @Shared char[] maxLength
   @Shared char[] tooLong

   def setupSpec() {
      maxLength = ["a"]*MAX_CHARACTERS
      tooLong = ["a"]*(MAX_CHARACTERS+1)
   }

   @Unroll("validating name '#name' should be #result as valid")
   def "name validator" () {
      given: "a new name validator"
         def nameValidator = new NameValidator();

      when: "given #name name is validated"
         def nameIsValid = nameValidator.isValid(name, null)

      then: "name should be #result as valid"
         nameIsValid == resultBoolean
      and: "validator should have produced #message error message #actualMessage"
         nameValidator.errorMessage == actualMessage

      where:
      name                                    | result     | resultBoolean  | message  | actualMessage
      "Valid name"                            | "accepted" | true           | "no"     | MSG_VALID
      "@ Should work"                         | "accepted" | true           | "no"     | MSG_VALID
      "1 Should work"                         | "accepted" | true           | "no"     | MSG_VALID
      new String(maxLength)                   | "accepted" | true           | "no"     | MSG_VALID
      "aa"                                    | "accepted" | true           | "no"     | MSG_VALID
      "Valid special chars  ÃÃÃÃÃ¶Ã_ -@./\\(" | "accepted" | true           | "no"     | MSG_VALID
      null                                    | "rejected" | false          | "one"    | MSG_INVALID_LENGTH
      ""                                      | "rejected" | false          | "one"    | MSG_INVALID_LENGTH
      "a"                                     | "rejected" | false          | "one"    | MSG_INVALID_LENGTH
      new String(tooLong)                     | "rejected" | false          | "one"    | MSG_INVALID_LENGTH
      "Invalid character!"                    | "rejected" | false          | "one"    | MSG_INVALID_CHAR
      "a ok but ^ is invalid character"       | "rejected" | false          | "one"    | MSG_INVALID_CHAR
      "@ ok but ? is invalid character"       | "rejected" | false          | "one"    | MSG_INVALID_CHAR
      "- is invalid start character"          | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      ". is invalid start character"          | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      "\\ is invalid start character"         | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      "/ is invalid start character"          | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      "( is invalid start character"          | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      ") is invalid start character"          | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      "& is invalid start character"          | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      "\' is invalid start character"         | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
      "+ is invalid start character"          | "rejected" | false          | "one"    | MSG_INVALID_FIRST_CHAR
   }
}
```

Figure 5.9: Figure 5.7 *JUnit* example refactored into *Spock* DDT feature method



```
▼ NameValidatorSpec                                                                            81ms
      validating name 'Valid name' should be accepted as valid                                  60ms
      validating name '@ Should work' should be accepted as valid                                1ms
      validating name '1 Should work' should be accepted as valid                                0ms
      validating name 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa                              0ms
      validating name 'aa' should be accepted as valid                                           0ms
      validating name 'Valid special chars åÅäÄöÖ_-@./\()&'+1' should be accepte                 0ms
      validating name 'null' should be rejected as valid                                         1ms
      validating name '' should be rejected as valid                                            10ms
      validating name 'a' should be rejected as valid                                            1ms
      validating name 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa                              0ms
      validating name 'Invalid character!' should be rejected as valid                           0ms
      validating name 'a ok but ^ is invalid character' should be rejected as valid              0ms
      validating name '@ ok but ? is invalid character' should be rejected as valid              1ms
      validating name '- is invalid start character' should be rejected as valid                 1ms
      validating name '. is invalid start character' should be rejected as valid                 0ms
      validating name '\ is invalid start character' should be rejected as valid                 3ms
      validating name '/ is invalid start character' should be rejected as valid                 0ms
      validating name '( is invalid start character' should be rejected as valid                 0ms
      validating name ') is invalid start character' should be rejected as valid                 1ms
      validating name '& is invalid start character' should be rejected as valid                 0ms
      validating name '' is invalid start character' should be rejected as valid                 1ms
      validating name '+ is invalid start character' should be rejected as valid                 1ms
```

Figure 5.10: *Spock* DDT feature method run output

Compared to JUnit, *Spock* and its DDT feature enabled more readable and concise test structure together with more information containing run output. At the time of research, project B had only one developer working with *JUnit* testing and he was convinced to take *Spock* into use after the illustrated refactored DDT example. *Spock* was used in new unit and integration level test classes while keeping the old *JUnit* tests intact. This happened without any major problems with the used build tool *Maven*.

Chapter 6

Results and discussion

This chapter covers the results and analysis of the case study with discussion to related research. First the results of interview for demographic purposes are presented. Second, survey and interview results regarding *JUnit* compared to *Spock* and *Spectrum* are analyzed. Third, the test code analysis is presented.

6.1 First interview: demographics and projects

First interview was conducted to study about the demographics of participants and projects under research. The demographics of participants are displayed in table 6.1. To summarize the participants, all can be categorized as senior software developers with many years of working with *xUnit family* testing frameworks. Only participant C has a fair amount of prior experience working with BDD-testing frameworks.

Participant attribute	Participant A	Participant B	Participant C
Project	Project A	Project A	Project B
Software development experience	17 years	15 years	8 years
Java development experience	14 years	1 year	5-6 years
Spring Framework experience	7 years	1 year	5-6 years
Automated unit testing experience *with frameworks*	14 years; *JUnit*	10 years; *CPPUnit, JUnit*	3-4 years; *JUnit, TestNG*
Automated integration testing experience *with frameworks*	14 years; *JUnit, Some Robot Framework with Selenium*	Hardly at all; *JUnit*	5-6 years; *JUnit, TestNG*
BDD-testing framework experience *with frameworks*	< 1 year; *Jasmine, Mocha*	-	2-3 years; *JBehave*

Table 6.1: Participant demographics

The projects under study can both be categorized as web application projects with Java *Spring Framework* backend technology. They both have an agile development process. **Project A** is a customer project in public administration context. **Project B** is an in-house project. They are both on the medium scale in code size, but financially project A can be categorized as medium to large category.

Project A has two developers, **participants A** and **B**, working with *Spring Framework* backend, where there is automated low-level testing in place with *JUnit*. Project B has only one backend developer, **participant C**, working with *Spring Framework* . As Project B was developed in larger scale from 2012 to 2013 and published originally in 2013, it has some of the automated low-level testing with *JUnit* done by others than participant C. Still most of this kind of test work is done by participant C. Table 6.2 displays the details of projects A and B.

Project attribute	**Project A**	**Project B**
Description	Web application for area management, replacing existing system	Wep application for working hours tracking & reporting
Context	Public administration, development for customer	In-house develoment
Development process	Agile development with customized *Scrum*	Agile with loosely defined process
Size & development team	Medium - large project; 3 developers for approximately 1,5 years	Small - medium project; Published 2013, now 2 developers maintaining & further development
Architecture	Client rendered single-page application	Server MVC with some client rendered views
Technologies	*Java Spring Boot & Angular 2*	*Java Spring Framework & JSP, Backbone.js*
Quality assurance process	Automated unit & integration testing, code reviews, continuous integration, no dedicated tester	Automated unit, integration & acceptance testing, code reviews, continuous integration, no dedicated tester
Used unit testing framework	*JUnit*	*JUnit*
Used integration testing framework	*JUnit* extended for *Spring Framework*	*JUnit* extended for *Spring Framework*
Chosen BDD-testing framework	*Spectrum*	*Spock*

Table 6.2: Project details

6.2 Surveys and BDD-testing framework feedback interviews analyzed

This section covers the results analyzed for both *JUnit* and BDD framework surveys together with participant BDD framework feedback interviews. First the developer practices and their changes are answered with the data. Second, developer perception of low-level testing and its changes are illustrated. Together these sections answer RQ1 & and RQ2.

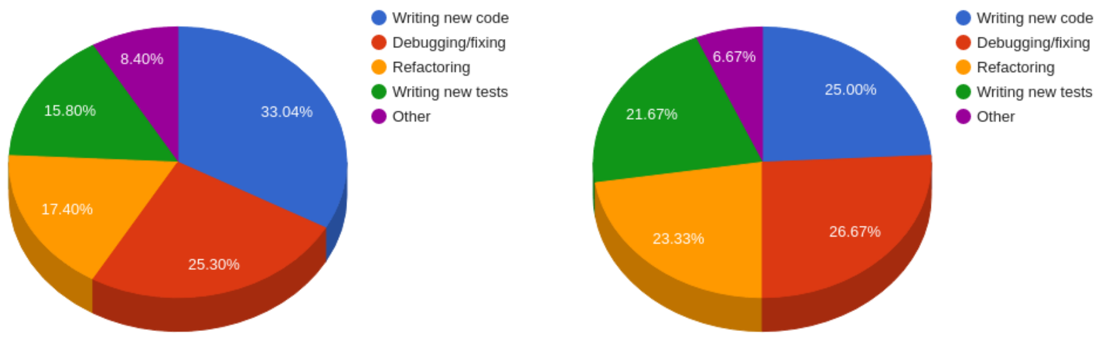
The baseline for answers was conducted with the survey described in detail in Appendix B section B.1. These questions related to *JUnit* low-level testing are displayed with **Q1, Q2, Q3** and so on. Their counterparts, questions aimed to study the changes in low-level testing with new BDD implementation level testing framework are marked as **Q1', Q2', Q3'** and so on. Details of BDD surveys can be found in Appendix B section B.2. Participants whom answered the surveys were A, B and C. The color coding of participants is visible throughout the result tables for easier highlighting of answers.

6.2.1 Automated low-level testing developer practices

This section answer the **RQ1:** *How does behavior-driven testing frameworks change developer practices working with automated low-level testing compared to JUnit framework?* This is studied through multiple aspects, which all display results and analysis individually. First, the developer software development & testing time and effort usage is analyzed. Second, low-level test optimizing targets are inspected. Third, test understandability and informativiness is under study. After that, test refactoring techniques and commenting practices are analyzed. Finally, unit testing practices are studied in more detail before summarizing the answers to RQ1 and the changes in low-level testing developer practices.

6.2.1.1 Software development & testing time and effort usage

First studied aspect of developer practices was **time usage**. Figure 6.1 displays the developer software development time usage in original study [5] and in this thesis. Table 6.3 display the participant values also individually and the changes in time usage after the introduction of a new BDD framework. In the original study, writing new code is the dominating activity in time usage with 33.04% share. At the beginning of this study when studying *JUnit* practices, participants used most of their software development time in *debugging or fixing* activities with 26.67% time usage share. *Writing new code* (25.00%) and *refactoring* (23.33%) were close by in time usage. The participants in this study had quite much variance in their



((a)) Results in original study [5] ((b)) Results in this study

Figure 6.1: Developer software development time usage

answers for time usage, but *writing new tests* was uniformly higher in time usage than in the original study results. Participants A and C were closely together in their answers for time usage, whereas participant B had more emphasis on initial *writing of new code* and less time spent on *refactoring*.

Question	**Answer options**						
Q1: How do you spend your software development time (in percentages)	Participant A	Participant B	Participant C	Average			
1. Writing new code	20%	40%	15%	25%			
2. Writing new tests	20%	25%	20%	21.67%			
3. Debugging/fixing	30%	25%	25%	26.67%			
4. Refactoring	20%	10%	30%	23.33%			
5. Other	10%	0%	10%	6.67%			
Q1': Compared to JUnit, How do you spend your software development time?	A lot less time	Less time	Slightly less time	The Same amount of time	Slightly more time	More time	A lot more time
1. Writing new code				A B C			
2. Writing new tests		C		A	B		
3. Debugging/fixing				A B C			
4. Refactoring			C	A B			
5. Other				A B C			

Table 6.3: Development time usage and changes in it

Changes in software development time were studied with the **Q1': ***Compared to JUnit, How do you spend your software development time?* Results are displayed in table 6.3. Participant A didn't see any changes in the overall software development time usages after introduction of *Spectrum*. This has to be taken with a grain of salt, as the next questions shows increase in testing efforts when comparing *JUnit* and *Spectrum*. Participant B noticed a slight increase in time used to write new tests with *Spectrum*, whereas other time usages remained the same. On the other hand, participant C said to use less time writing new tests and slightly less time refactoring the code with *Spock*. These testing time usages are analyzed in detail with Q2'.

Questions **Q2** and **Q2'** further analyze the time usage, now focusing in automated testing. Their results are shown in table 6.4. All participants answered to use approximately 30 minutes of time in single test case. The averages show that about one third (28.33%) of initial effort goes to *thinking* about the test case without actual implementation. About two thirds (71.67%) of initial effort goes to *implementation* of the test case. *Refactoring* takes about one third (28.33%) of the overall testing effort.

Participants had highly varying answers in initial *thinking* and *implementation* effort and also in overall testing *refactoring* effort. Participants B and C had same kind of profile in initial efforts, where about one fifth of it goes to initial *thinking* of test case and four fifths to *implementation*. Participant A had the initial effort going in half between the two aspects. In overall effort, *refactoring* of test code took 50% of effort for participant A. Participant B hardly at all *refactored* test code (5%) and participant C used around one third of overall testing effort to *refactoring* (30%).

Question	**Answer options**						
Q2: How do you spend your low-level automated testing time	Participant A	Participant B	Participant C	Average			
1. How much approximately you use time per test case (minutes)?	30 min	30 min	30 min	30 min			
2. How much of your initial effort goes to thinking about test case content without implementation?	50%	20%	15%	28.33%			
3. How much of your initial effort goes to initial test case structuring and implementation?	50%	80%	85%	71.67%			
4. How much of your overall testing effort goes to refactoring test code (percentage)?	50%	5%	30%	28.33%			
Q2': Compared to JUnit, How do you spend your low-level automated testing time?	A lot less	Less	Slightly less	The Same amount	Slightly more	More	A lot more
1. Do you use more or less time per test case?		C		A	B		
2. Do you use more or less of initial effort thinking about test case content?				C	A B		
3. Do you use more or less of initial effort to test case structuring and implementation?				C	B	A	
4. Do you use more or less of overall testing effort to refactoring test code?			C	A	B		

Table 6.4: Development & testing time and effort usage and changes in them

Changes in automated testing time and effort were studied with **Q2':** *Compared to JUnit, How do you spend your low-level automated testing time?* Participant A answered to use around the same time per test case with *Spectrum*, but also answered to use slightly more effort in initial *thinking* of test case and more effort in test case *implementation.* The overall test *refactoring* effort was the same amount. All effort changes combined, it might be feasible to think that the time per test case might have increased. In the feedback interview of *Spectrum* with participant A, he said that the implementation and how to effectively structure the usage of Java 8 lambdas and *Spectrum* variables in them took extra time compared to *JUnit.* Participant B answered to use slightly more time with *Spectrum* on all aspects of low-level testing. In the followed interview, this was found out to be caused by in general the new testing framework and especially the lambda structure of tests with it. Participant C answered to use less time per test case with slightly less effort on test refactoring. In the following interview, participant C answered that it was quick to feel effective with *Spock* and especially its DDT allowed to easily test out different situations.

In conclusion, *Spectrum* seems to have a longer learning curve than *Spock.* Participants A and B both said in interview that it still feels like there is learning to do to be effective with *Spectrum* . Participant C had previously used JBehave which also uses the same *Gherkin*-structure as *Spock*, therefore it might have eased the introduction of *Spock* for him. Survey results also support this. After two months, testing with *Spectrum* seems to take more effort and time than with *JUnit.*

6.2.1.2 Test optimizing targets

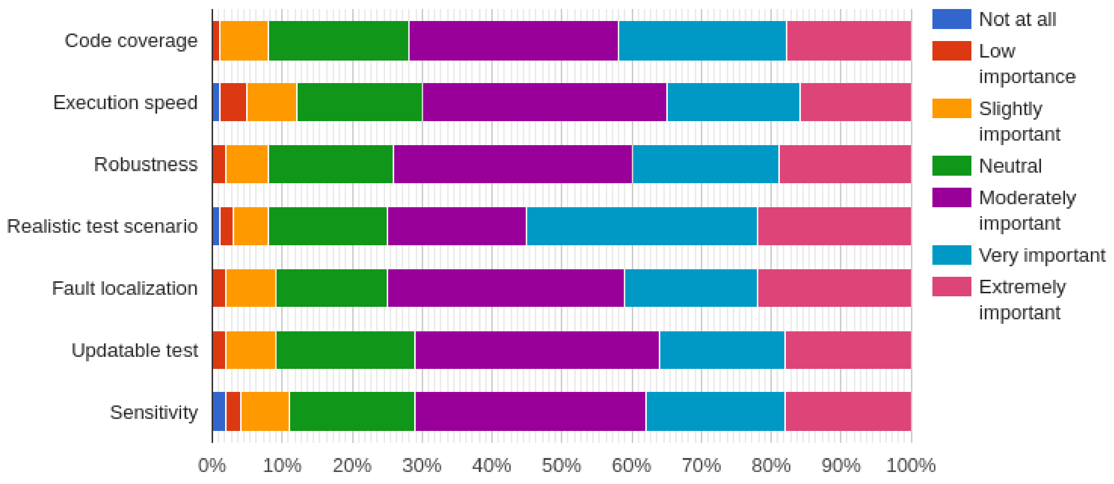


Figure 6.2: Original study [5] unit test optimizing target percentages amongst developers

Original study by Daka and Fraser [5] studied unit testing **optimizing targets** for tests. Figure 6.2 displays the results from original study. All the studied aspects had at least over 60% of developers finding them ranging from moderately important to extremely important. *Realistic test scenario* was the most important optimizing target amongst original study survey participants.

Regarding *JUnit* low-level testing, participants in this study had varying importance in studied aspects. Full answers are displayed in table 6.5 with **Q3:** *How important are the following aspects for you when you write new low-level tests?* Participants A and B answered the most alike, whereas participant C had slightly different optimizing profile. *Code coverage* was a neutral optimizing target for all participants. Additional question, original to this study, was the *capturing of behavior* as optimizing target. Participants A and B found it neutral and slightly important, but participant C had it as a very important optimizing target. Another interesting target was *execution speed*, where participants A and B had it at least moderately important, but participant C answered it as low importance optimizing target. *How realistic the test scenario is* was very important to optimize for participant C. This was also the most important target amongst original study survey participants. For participants A and B, *realistic test scenario* was only slightly important. These kind of variations in optimizing targets are quite natural when studying this low participant count. The original study figure 6.2

displays how all the targets are quite close to each other with larger sampling.

Question	**Answer options**						
Q3: How important are the following aspects for you when you write new low-level tests?	Not at all	Low importance	Slightly important	Neutral	Moderately important	Very important	Extremely important
1. Code coverage				A B C			
2. Capturing all behavior of unit/feature with tests or assertions			B	A		C	
3. Execution speed		C			B	A	
4. Robustness against code changes (i.e., test does not break easily)					A B C		
5. How realistic the test scenario is			A B			C	
6. How easily faults can be localised/debugged if the test fails			B		A C		
7. How easily the test can be updated when the underlying code changes		C	A		B		
8. Sensitivity against code changes (i.e., test should detect even small code changes)		A B		C			
Q3': Compared to JUnit, How important are the following aspects for you when you write new low-level tests?	A lot less important	Less important	Slightly less important	As important as before	Slightly more important	More important	A lot more important
1. Code coverage				A B C			
2. Capturing all behavior of unit/feature with tests or assertions				B C	A		
3. Execution speed				A B C			
4. Robustness against code changes (i.e., test does not break easily)				A B C			
5. How realistic the test scenario is				A B		C	
6. How easily faults can be localised/debugged if the test fails				B	A C		
7. How easily the test can be updated when the underlying code changes				A B C			
8. Sensitivity against code changes (i.e., test should detect even small code changes)				A B C			

Table 6.5: Optimizing targets in low-level tests and changes in them

The changes in low-level testing optimizing targets were studied with **Q3': *Compared to JUnit, how important are the following aspects for you when you write new low-level tests?*** Participant A compared the changes from *JUnit* to *Spectrum* and find most of the targets as important as before. *Capturing behavior* of tested subject in tests was slightly more important than before. Also *fault localization* was slightly more important target than before. These two targets and their higher importance seem natural for a *xSpec family* testing framework, as it aims to promote more granular and more natural language description information holding test cases than *JUnit*. Participant B didn't feel there to be any optimizing target changes. In the interview he stated that none of the targets really changed one way or another for him when using *Spectrum* instead of *JUnit*.

Participant C found *realistic test scenario* to be more important with *Spock* than it was with *JUnit*. Realistic test scenario was already very important for him with *JUnit* and with *Spock*, it is even more important. He also answered *easy fault localization* to be slightly more important than before. In the interview it was found out that these aspects had more importance because of the BDD-style of writing the tests with *Spock*. For participant C, *capturing behavior* with tests was already very important with *JUnit* and it was interesting that *Spock* didn't change this. The *COTM*-metric with *JUnit* in test code analysis section shows support for already granular unit test cases in project B. This might act as an indicator of already behaviorally described test methods.

To conclude, its hard to see any pattern in test optimizing target changes. Altogether, the changes were quite mild. Easier fault localization was the only common nominator that was found slightly more important with two out of three participants. Therefore it might be feasible to say, that test optimizing targets aren't too tied to the testing framework in this study context. This might show different results with larger sampling repeating the same kind of study. The results might be different especially when using these implementation level BDD-testing frameworks for actual practicing of BDD. Then the optimizing targets might be much different than with *xUnit family* testing.

6.2.1.3 Test understandability and informativeness

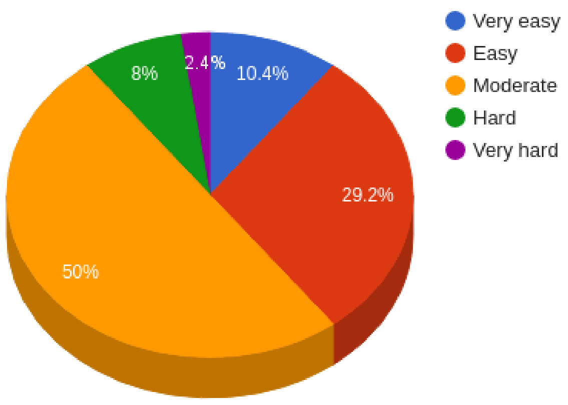


Figure 6.3: Original study [46] unit test understandability

The difficulty of developers **understanding** unit tests was studied by Li et al. [46]. Over 60% of survey participants found it at least moderately difficulty. Figure 6.3 illustrates the total survey answer regarding unit test understandability. This question was chosen to be replicated in this study to see, if the research hypothesis of developers finding it easier to understand test cases with BDD-testing frameworks will hold true.

Table 6.6 summarizes this study participant answers. First, **Q4** studied how difficult understanding of low-level tests with *JUnit* was. The results mirrored quite well the original study findings, as participants B and C find the understandability to be moderately difficult and participant A found it hard to understand a low-level test.

Question	**Answer options**						
	Very easy	Easy	Moderate	Hard	Very hard		
Q4: How difficult is it for you to understand a low-level test?			B C	A			
	A lot less difficult	Less difficult	Slightly less difficult	As difficult as before	Slightly more difficult	More difficult	A lot more difficult
Q4': Compared to JUnit, how difficult is it for you to understand a low-level test?	C	A			B		

Table 6.6: Understandability of low-level tests and changes in it

The changes in low-level test understandability were studied with **Q4': ***Compared to JUnit, how difficult is it for you to understand a low-level test?* Participant A found it less difficult to understand a low-level test after the introduction of *Spectrum*. Participant B find it slightly more difficult to understand a low-level test. This was further analyzed with interview, where he says that the divided nested structure of *Spectrum* files make it slightly harder to understand individual tests as whole. Compared to *JUnit*, Participant C found it a lot less difficult to understand *Spock* tests. In the interview this was further inspected and he states that its much more easier to write information to *Spock* feature methods compared to *JUnit* test methods.

Summing up, participant surveys and interviews seem to support partially the **H2:** *"Developers will find it easier to understand test cases"*. Although, participant B find it slightly harder to understand low-level tests at times, there is still more evidence for easier undestanding. H2 is further studied with following questions in this section.

Table 6.7 displays questions **Q5 & Q5'** and their results. Q5 was chosen to see how easily the *JUnit* test methods can be **structured** and **understood** to hold different parts of the test. *JUnit* testing literature states that the structure should be done with *Arrange-Act-Assert* [23], but it is not enforced and thus can lead to poorly structured tests which are hard to understand [24]. Q5 studies if this does hold true.

All the participants had different parts of test structure with ranging difficulties in structuring and understanding. Participant A could be categorized to hold almost all parts of structuring the test somewhat difficult, with the exception of finding it easy to structure information to *context* of tests. Reading tests and understanding their structure was found at least from slightly hard to hard. Participant B find writing information to *assertions* of test easy, but otherwise structuring the test was found from moderately difficult to slightly hard. Reading the test structure parts for information was equivalent to writing and structuring of tests. Participant C found it somewhat difficult to produce information to tests, but reading the test structure was found to be slightly easy. These ranging results are not too surprising, as *JUnit* does not promote a certain structure for tests. It seems to result in hard to produce stucture that is not too easy to understand.

Question	**Answer options**						
Q5: In low-level testing, how difficult is it for you to	Very easy	Easy	Slightly easy	Moderate	Slightly hard	Hard	Very hard
1. Structure and write information to context of test?		A			B C		
2. Structure and write information to stimulus of test?				A B C			
3. Structure and write information to assertions of test?		B	C		A		
4. Read test case structure for information about context of test?			C		B	A	
5. Read test case structure for information about stimulus of test?			C	B	A		
6. Read test case structure for information about assertions of test?		B	C			A	
Q5': Compared to JUnit in low-level testing, how difficult is it for you to	A lot less difficult	Less difficult	Slightly less difficult	As difficult as before	Slightly more difficult	More difficult	A lot more difficult
1. Structure and write information to context of test?	A C		B				
2. Structure and write information to stimulus of test?		A C			B		
3. Structure and write information to assertions of test?		C	A	B			
4. Read test case structure for information about context of test?	C		A B				
5. Read test case structure for information about stimulus of test?		A C		B			
6. Read test case structure for information about assertions of test?		C	A B				

Table 6.7: Low-level test structure informativiness and changes in it

Q5' displays the changes in structuring and reading of tests for information. Participant A found *Spectrum* on all occasions easier to structure information to tests and also to read it from tests. Structuring information to tests could be categorized as quite much easier. Reading the tests for information was found somewhat easier than with *JUnit*. Participant B found the changes in structuring of tests with *Spectrum* to range from slightly less difficult to slightly more difficult, but the reading of test case structure could be summed up to be somewhat more easier than with *JUnit*. In the interview he said that the main benefits of *Spectrum* was the easier spotting of different parts of test. Participant C found it ranging from less to a lot less difficult structuring and reading the tests.

In conclusion, the results show that *structuring* and *reading* of the different parts of tests was found almost unanimously **more easier** with BDD-testing frameworks than with *JUnit*. This finding supports the **H2**, but for more evidence, the next questions Q6 and Q6' need to be analyzed.

Table 6.8 displays questions **Q6 & Q6'** with results. First Q6 studies how **informative** participants found *JUnit* test output. Participant A find the output hardly informative, whereas participants B and C found the output ranging from somewhat informative to moderately informative. These answers could be a result from typical *JUnit* test method naming, which doesn't tend hold that many words. Section 6.3.2 displays that the average word count in test methods in projects were ranging from approximately 5 to 6. This amount of words can't hold naturally too much information in total.

Question	**Answer options**						
	Not at all	Hardly informative	Slightly informative	Somewhat informative	Moderately informative	Very informative	Extremely informative
Q6: How informative you usually find the test output?		A		C	B		
	A lot less informative	Less informative	Slightly less informative	As informative as before	Slightly more informative	More informative	A lot more informative
Q6': Compared to JUnit, how informative you usually find the test output?				B	A	C	

Table 6.8: Low-level test output informativeness and changes in it

Q6' studied the change of test output informativeness after the introduction of new BDD-testing framework. Participant A found a slight increase in informativeness of output. Participant B found the output as informative as before, whereas participant C found it more informative. As the *xSpec family* testing frameworks allow free text description to their code example groups and code examples, the answer of participant B was further studied in interview. He stated that he usually only checks from the output whether test passes or fails and as such doesn't see any improvement when comparing *Spectrum* and *JUnit*.

On the whole, it seems that **H2:** *"Developers will find it easier to understand test cases"* has strong evidence to support it. However, the results of questions Q4'-Q6' are not clear-cut unanimous and especially the nested structure of **xSpec family** code example groups might not be so clear at the beginning. In any case, it can be concluded that BDD-testing frameworks allow to structure the tests for separate parts, which helps in identifying *context, action* and *assertions* more explicitly from tests.

6.2.1.4 Test code repetition reducing techniques

Table 6.9 displays the usage of different **repetition reducing** techniques. First **Q7:** *How much are the following repetition reducing techniques used in your low-level testing?* studies how much a set of common refactoring techniques stated in unit testing literature [21] are used in projects.

Most common used refactoring techniques amongst survey participants were *extract method* and lifecycle hook *before -each.* These two techniques were in use frequently or very frequently within participant testing. Lifecycle hook *before -class* was also in use at least occasionally. Automatic test generation through DDT was found very rarely or rarely used with participants A and B. Section 6.3.2 shows that in project A it was not used at all, where data-driven test method ratio to standard low level test methods was 0. Participant C answered to use occasionally DDT and the test code analysis data in section 6.3.2 supported this. Common test initializer class inheritance was a practice very rarely or rarely used amongst all participants. During interview, participant A even stated test class inheritance to be *"a practice to avoid, it's devil's work"*.

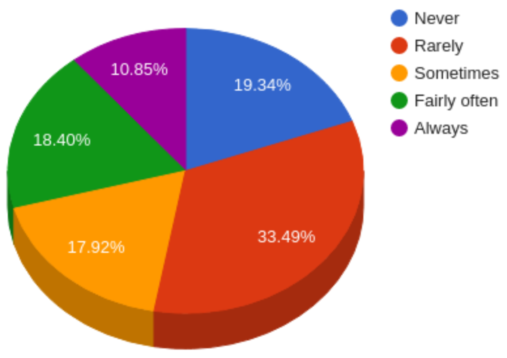
Question	**Answer options**						
Q7: How much are the following repetition reducing techniques used in your low-level testing?	Never	Very rarely	Rarely	Occasionally	Frequently	Very frequently	Always
1. Extract method (custom helper methods)					A B	C	
2. Lifecycle hooks Before/After (class)				C	A B		
3. Lifecycle hooks Before/After (each)					A B	C	
4. Automatic test generation via test method parametrization		A	B	C			
5. Common test initializer class inheritance		A C	B				
Q7': Compared to JUnit, how much are the following repetition reducing techniques used in your low-level testing?	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
1. Extract method (custom helper methods)			B	A C			
2. Lifecycle hooks Before/After (class/all)				A B C			
3. Lifecycle hooks Before/After (each)				B C		A	
4. Automatic test generation via test method parametrization				A B			C
5. Common test initializer class inheritance				A B C			

Table 6.9: Used repetition reducing techniques for low-level testing and changes in their use

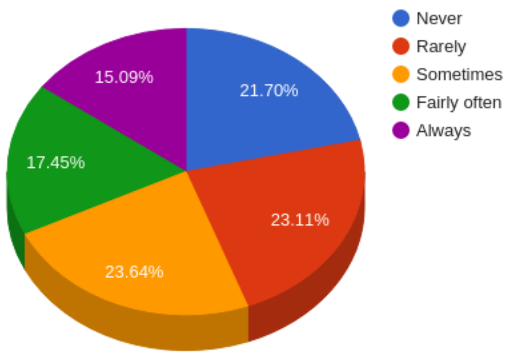
Q7': *Compared to JUnit, how much are the following repetition reducing techniques used in your low-level testing?* studies how techniques are used differently with the new BDD-testing framework to reduce repetition. Participant A said to use techniques the same amount as before, with the exception of lifecycle hook *before -each* being used more. In the feedback interview, participant A stated that there was some confusion when to use *before -all* lifecycle hook and therefore hook *before -each* was mainly used. The increased use of *before -each* is explained by the nested code example groups, which each support their own separate lifecycle hooks. Participant B used the repetition reducing techniques the same amount, except for slightly less used *extract method*. Participant C answered to use *automatic test generation via test method parametrization* (DDT) a lot more than before. Test code analysis in section 6.3.2 supports this claim.

In conclusion, both BDD-testing frameworks allow new ways to reduce repetition in test code. How much they are used seems to be independent on the developer. Later is studied how much more maintainable these techniques make the test code in eyes of the participants.

6.2.1.5 Test commenting practices



((a)) Comment adding



((b)) Comment updating

Figure 6.4: Original study [46] unit testing commenting practices

Test commenting and **comment updating** was found beneficial, but also a practice rarely done by most of the developers [46]. Figure 6.4 shows the original study results regarding test commenting practices. Participants in this study seem to divide with commenting practices. Table 6.10 displays the participant *JUnit* commenting practice results with **Q8** and **Q9**. Project A and its participants A and B answer fairly often to add and update comments in low-level test cases. Participant C rarely practices test commenting or updating. Test data analysis in

6.3.2 correlates with participant answers, as project A had quite much commenting in test methods and project B very little.

Question	**Answer options**						
	Never	Rarely	Sometimes	Fairly often	Always		
Q8: How often do you add/write documentation comments to low-level test cases?		C		A B			
	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
Q8': Compared to JUnit, how often do you add/write documentation comments to low-level test cases?				B C		A	
	Never	Rarely	Sometimes	Fairly often	Always		
Q9: When you make changes to low-level tests, how often do you comment the changes (or update existing comments)?		C		A B			
	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
Q9': Compared to JUnit when you make changes to low-level tests, how often do you comment the changes (or update existing comments)?				B C	A		

Table 6.10: Documentation practices in low-level testing and changes in them

Questions **Q8'** and **Q9'** in table 6.10 study the changes in commenting with new BDD-testing framework. Participant A answers to add more comments with *Spectrum* to low-level test cases than before and also to update the comments slightly more. Here the test code analysis data in section 6.3.2 doesn't support this directly, as pure comments in test methods have decreased drastically. But overall the test contains more textual description in use with code example group and code example descriptions, which can be seen as increased test method name word count. Participant B answered that he adds and updates the comments with *Spectrum* the same amount as before. This is also in contradiction with the test code analysis *COC*-metric change in project A. Participant C said to add and update the comments with *Spock* the same amount as before. Test code analysis supports this.

In conclusion, there seems to be a high possibility that Q8' and Q9' were not understood by all participants in the manner that their original intent was. The actual changes in commenting practices are examined in more detail in test code analysis section 6.3.2.

6.2.1.6 Unit testing practices

Pure unit testing practices were also studied with the participant surveys. First the **granularity** of unit tests were studied with questions **Q10** and **Q10’** displayed in table 6.11. Participants A and B answered to write approximately two to three tests methods per class method that contain few assertions inside them each. Participant C answered to produce more granular test cases, where there exists four to five test methods per class method. Also the assertion count was stated to be higher, 4-5 assertions per test method. Test data analysis in section 6.3.1 seems to support quite well all these participant answers, although the assertion count per test method calculated in section 6.3.2 for project B was slightly lower than the answered four to five assertions per test method.

Question	**Answer options**						
Q10: In unit testing, how many		1	2-3	4-5	6-7	8-9	10 or more
1. Test methods do you usually write per class method?			A B	C			
2. Assertions do you usually write per test method?			A B	C			
Q10’: Compared to JUnit in unit testing	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
1. Do you write more or less test methods per class method?					B	A C	
2. Do you write more or less assertions per test method?		A	B	C			

Table 6.11: Unit testing practices and changes in them

Changes in unit testing granularity, after the introduction of BDD-testing framework, were studied with Q10’. Participant A answered that with *Spectrum* he wrote more test methods per class method with less assertion inside individual test methods. Participant B used also *Spectrum* and he felt that he wrote slightly more test methods per class method and these test methods contained slightly less assertions in them. Participant C answered the question for *Spock*. He felt that he wrote more test methods per class method with same amount of assertions in them. In the interviews, both participants A and B said that *Spectrum* promotes to write more granular tests. Participant A also felt that this is one of the main benefits of *Spectrum* over *JUnit*.

Summing up, it can be said that developers feel that they write more granular unit test cases with more test methods in them. Test code analysis in sections

6.3.1 and 6.3.2 are quite good in sync with the participant answers. Results from Q10 & Q10' and interviews support the **H1:** *Developer will write more granular test cases.*

Next studied unit testing practice is **isolation**. In related research, survey participant developers found unit test isolation to be a demanding practice [5]. Therefore questions **Q11, Q11', Q12** and **Q12'** displayed in table 6.12 are aimed to learn more about isolation in unit testing. First survey question Q11 asks what mocking library is normally used in *JUnit* unit test isolation. After that, Q12 studies wich aspects of isolating are find hard.

Usually unit test isolation was done with *Mockito* by all participants. Participant C is the only one that found all aspects of unit test isolation to be easy or slightly easy. Participant A found isolation usually slightly easy with *verifying* of mock object actions being moderate in difficulty. Participant B found most of isolation easy, but *stubbing* was considered slightly hard. Altogether, isolation activities in *JUnit* unit testing wasn't found too demanding by any of the participants.

Question	**Answer options**						
	Mockito	jMock	Powermock	Easymock	Other		
Q11: In unit testing, what mocking library do you normally use?	A B C						
	Mockito	jMock	Powermock	Easymock	Spock's internal mocking	Other	
Q11': In unit testing with Spectrum/Spock, what mocking library do you normally use?	A B C						
Q12: In unit testing, how difficult you find it to	Very easy	Easy	Slightly easy	Moderate	Slightly hard	Hard	Very hard
1. Mock objects?		B C	A				
2. Stub method calls?			A C		B		
3. Verify mock object actions?		B C		A			
Q12': Compared to JUnit in unit testing, how difficult you find it to	A lot easier	Easier	Slightly easier	As difficult as before	Slightly harder	Harder	A lot harder
1. Mock objects?				B C	A		
2. Stub method calls?				B C	A		
3. Verify mock object actions?				B C	A		

Table 6.12: Unit testing practices and changes in them

After the introduction of *Spectrum* in project A, the mocking library stayed intact, as *Spectrum* is a Java-based custom runner for *JUnit*. Although the mocking library was the same, participant A found all aspects on unit test isolation slightly harder than before. This was further analyzed in interview with participant, where he states that the usage of *Mockito*-functions inside **lifecycle hooks** and **block lambdas** was not as straightforward as with *JUnit* tests. Participant B didn't feel any change in difficulty of usage of *Mockito* when using it with *Spectrum* instead of *JUnit*. Participant C answered to still use *Mockito* with *Spock* and this was interesting. Later when analyzing the test code, I found out that due to low number of new unit tests, no mocking was needed in them. As seen in earlier chapters, *Spock* provides its internal mocking. Unfortunately within this time period of study, no comparison could be made between *Mockito* and *Spock*'s internal mocking. In brief, it could be said that for unit test *isolation*, there might exist a learning curve with *Spectrum* and usage of *Mockito* with it.

Question	**Answer options**		
	Before implementation	During implementation	After implementation
Q13: When do you add automated unit tests for developed code with JUnit?		B C	A
	Before implementation	During implementation	After implementation
Q13': When do you add automated unit tests for developed code with Spectrum/Spock?		B C	A

Table 6.13: Unit testing practices and changes in them

Questions **Q13** and **Q13'** were aimed to study when are unit tests add for code with *JUnit* and the new BDD-testing framework. The main idea was to see, does the introduction of BDD-testing framework by itself promote test first -principle without additional changes in testing. For participant A there was no change, as the unit tests were added after the implementation with both *JUnit* and *Spectrum*. Participants B and C experienced also no change in when the tests are added for code. Both said to add unit tests during implementation.

It seems that although implementation level BDD-testing frameworks were originally designed to help with test first -principle, they don't seem to promote this in study context. Still, these frameworks do seem to work quite well within the studied aspects of low-level testing without the use of BDD.

6.2.1.7 Change summary

Benefit	Participants agreeing
Writing more granular test cases	A, B, C
Less time used in testing	C
Easier to understand test	A, C
Easier to structure test for different parts	A, C
Easier to read test for different parts	A, B, C
More informative test output	A, C
Less repetition through lifecycle hooks	A
Easier data-driven testing	C

Drawback	
More time and effort used in testing	A, B
Slightly harder to understand test	B
Slightly harder to isolate unit tests	A

Table 6.14: Developer changes in low-level testing practices

To answer **RQ1**, table 6.14 displays the summary of changes in low-level testing practices with new BDD test frameworks. There were lot of practices where at least two out of third participants felt improvement over *JUnit*. Some parts of studied aspects of low-level testing remained unchanged after the introduction of BDD-testing framework. There were only few parts where practice changes with BDD-testing frameworks resulted in drawbacks compared to *JUnit*. More specific, only *Spectrum* was found problematic at times. Main findings to RQ1 are:

In low level testing, BDD-testing frameworks change developers to write more granular test cases. Test context, action and assertions are easier to identify from the test than before. There is also evidence supporting overall easier understanding of tests with more informative test output. Repetition can also potentially be reduced more easily through different framework features.

Changing to xSpec family test framework Spectrum introduces a steeper learning curve than Gherkin family Spock. At the beginning, unit test isolation and understanding of tests can potentially be harder with Spectrum than with JUnit.

6.2.2 Developer perception towards automated low-level testing

This section answer the **RQ2:** *How does behavior-driven testing frameworks change developer perception of working with automated low-level testing compared to JUnit framework?* First, developer perception of automated low-level testing and changes in it are analyzed with survey and interviews. Second, developer loyalty towards low-level automated testing and specific testing frameworks are studied with **NPS** survey questions and separate interviews.

6.2.2.1 Developer perception of low-level testing

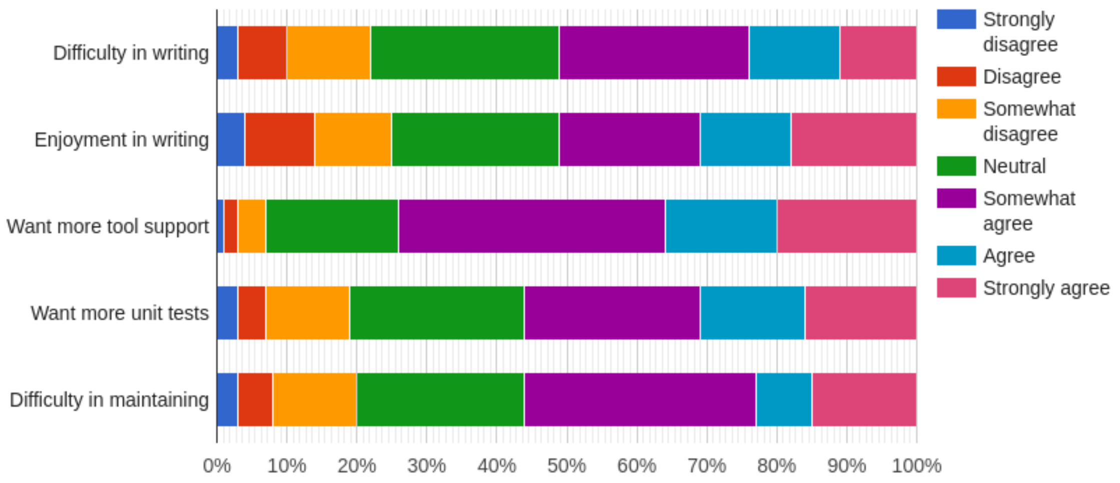
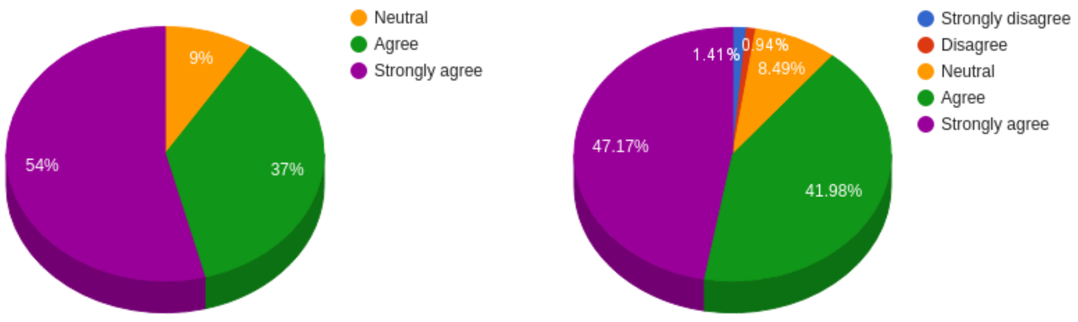


Figure 6.5: Original study [5] developer perception of unit testing

Daka and Fraser [5] have studied developer perception towards traditional unit testing. Interesting points that are further studied for low-level testing in this study are shown in figure 6.5. Some aspects to highlight are the facts that only about 50% of survey responders *enjoy writing* unit tests to some degree. Unit testing motivation was also found problematic in practitioner survey research by Runeson [6]. One of the hypothesis in this study was that implementation level BDD-testing frameworks should increase low-level testing enjoyment. Another interesting aspect is the *difficulty in maintaining* unit tests, where around 55% of study participants feel maintaining at least somewhat difficult. Runeson also discovered this same difficulty amongst survey participants [6]. One of the study hypothesis was that BDD-testing frameworks should make it less difficult to maintain code and especially the test code. Table 6.15 displays the **Q14**, which studies the developer perception towards low-level testing using Daka and Fraser's survey question as a base.

Q15a in table 6.15 studies the developer perception aspect further. Its questions have originally been used individually in two different studies [26, 46]. The original questions and their answers are displayed in figure 6.6. Interesting fact was that over 90% of original survey respondents feel that unit tests help in producing higher quality code than without them [26]. Around same 90% of different study participants feel that maintaining unit tests is important for system quality [46].



((a)) Unit tests help in producing higher quality code [26]

((b)) Maintaining unit tests is important for the quality of the system [46]

Figure 6.6: Developer perception of unit testing in original studies

Participants in this study, based on their perception towards low-level automated testing with *JUnit*, can be divided into two categories. Participants A and B are close to each other, they feel that *writing* low-level tests is somewhat difficult, *enjoyment* in writing is neutral or negative and *maintaining* of low-level tests is seen at least somewhat difficult. They both still have trust that low-level testing is *helpful in finding defects*, but **don't** feel that *JUnit promotes to write high quality test code.* Participants A and B seem to fall into same category as majority of respondents in the original research.

Participant C could be categorized into a testing oriented developer when inspecting perception towards low-level testing. He enjoys *writing* low-level tests and doesn't find it that difficult. He also doesn't find *maintaining* low-level tests too difficult. He thinks that low-level tests that he has written with *JUnit* will *help* others *understanding* the implemented production code. Like other participants, participant C also trusts that low-level tests will *help in finding defects*, but he **doesn't** feel that *JUnit promotes him to write high quality* test code.

Although all study participants feel that *JUnit* doesn't promote in writing higher quality test code, all of them still believe that low-level tests *help in producing higher quality code.* They all feel that *maintaining* low-level tests and their documentation is important for the system quality. All of the participants seem to follow the majority of developers in previous researches regarding test helping in quality and test maintaining importance. Table 6.15 displays the detailed answers of participants.

Question	**Answer options**						
Q14: Please indicate your level of agreement with the following statements	Strongly disagree	Disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Agree	Strongly agree
1. Writing low-level tests is difficult			C		A B		
2. I enjoy writing low-level tests		B		A		C	
3. I would like to have more tool support when writing low-level tests			B	A	C		
4. I would like to have more low-level tests				A B C			
5. Maintaining low-level tests is difficult			C		B	A	
6. I think my low-level tests will help other developers to understand the implemented unit/feature better			B		A	C	
7. Low-level automated testing helps me find defects in the code before other quality assurance phases					B	C	A
8. JUnit promotes me to write high quality test code			A C	B			
Q15a: Please indicate your level of agreement with the following statements	Strongly disagree	Disagree	Neutral	Agree	Strongly agree		
1. Overall, low-level tests help me produce higher quality code				B	A C		
2. Maintaining good low-level test cases and their documentations is important to the quality of a system				A B	C		

Table 6.15: Developer perception of low-level testing with *JUnit*

Table 6.16 display questions **Q14'** and **Q15a'** together with their results. With *Spectrum* there exists some controversial and interesting results. Participant A somewhat agrees that *writing* and *maintaining* of low-level tests with *Spectrum* is more difficult than with *JUnit*. Writing difficulty was further analyzed in the interview, where he states that the learning curve how to structure code example groups and code examples was still ongoing. Also what to write in *describe*-blocks was somewhat difficult. Slight increase in difficulty in maintaining low-level tests with *Spectrum* comes from the use of example groups properly, as participant A feels that it isn't always straightforward. In spite of slightly harder maintenance in total, he feels that it is easier to reduce repetition with *Spectrum* 's *lifecycle hooks*.

Although writing and maintenance were found slightly harder, participant A somewhat agrees that he *enjoys* writing low-level tests with *Spectrum* more than with *JUnit*. He also thinks that *Spectrum* allows other developers to *understand* the production code better than earlier with *JUnit*. Participant A feels that *Spectrum promotes him to write higher quality test code* than JUnit. In addition, he felt that *maintaining* of *Spectrum* tests was somewhat more important for system quality, but didn't see *Spectrum* helping in producing hiqher quality production code.

Question	**Answer options**						
Q14'&Q15a': Please indicate your level of agreement with the following statements	Strongly disagree	Disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Agree	Strongly agree
1. Writing low-level tests with *Spectrum/Spock* is more difficult than with JUnit		C			A	B	
2. I enjoy writing low-level tests with *Spectrum/Spock* more than I do with JUnit				B	A	C	
3. I would like to have more tool support for *Spectrum/Spock* when writing low-level tests				C		B	A
4. I would like to have more low-level tests with *Spectrum/Spock*				B	A C		
5. Maintaining low-level tests with *Spectrum/Spock* is more difficult than with JUnit		C		B	A		
6. I think my low-level tests with *Spectrum/Spock* will help other developers to understand the implemented unit/feature better than earlier tests with JUnit				B		A C	
7. Low-level automated testing with *Spectrum/Spock* helps me find defects in the code before other quality assurance phases better than earlier tests with JUnit			A B	C			
8. *Spectrum/Spock* promotes me to write higher quality test code than with JUnit		B				A C	
9. Overall, low-level tests with *Spectrum/Spock* help me produce higher quality code than with JUnit		B	A		C		
10. Maintaining good low-level test cases and their documentation with *Spectrum/Spock* is more important for system quality than maintaining JUnit test cases		B			A C		

Table 6.16: Developer perception changes in low-level testing with *Spectrum/Spock*

Participant B finds *writing* of low-level tests with *Spectrum* more difficult than with *JUnit*. This was further studied in the interview, where he said that quite much of the trouble in writing came from Java features used to write *Spectrum* tests, but also one cause was the more difficult structuring of tests. Both participants A and B agree or strongly agree that they would like to have *more tool support* for *Spectrum*. This was studied in following interviews, where both would like to see more IDE support. For most of the statements, participant B neither agrees nor disagrees. He disagrees with the statement that *Spectrum* promotes to write higher quality test code than *JUnit*. This was further discussed in the interview, where he says that he feels that the test case structure with *Spectrum* is more complex than with *JUnit*. Participant B disagreed with statements that *Spectrum helps producing higher quality production code* than *JUnit* or that *maintaining Spectrum* tests would be *more important* for system quality.

Participant C disagreed that *writing* and *maintaining* of low-level tests with *Spock* is more difficult than with *JUnit*. In the following interview, he stated that writing tests was quick to learn and they could be more easily divided into logical parts with *Given-When-Then* blocks. It is interesting to see that participant C initially already *enjoyed* writing low-level tests with *JUnit* and agreed to *enjoy* even more low-level testing with *Spock*. With *Spock*, he also feels that other developers will *understand* the implemented unit or feature better than with *JUnit*. Participant C agrees that all in all *Spock promotes* him to write higher quality test code. In total, participant C seemed to perceive *Spock* well, as he also somewhat agrees that *Spock helps to produce overall higher quality code* and that *maintaining* of *Spock* tests is more important for system quality.

Table 6.17 displays question **Q15b** and its answers. Q15b holds two sub-question statements that the participant can answer if they feel the same way or not. Sub-question 1 relates directly to hypothesis 2 and sub-question 2 to hypothesis 3. Participant A and C feel that they write both more *understandable* and more *maintainable* low-level tests with *Spectrum* and *Spock* than with *JUnit*. Participant B on the other hand feels neither sub-question statements to be true. In the feedback interview, these questions were further studied. Participant B feels that *Spectrum* has just a different way of doing things than *JUnit* and both have their areas where they perform better than the other.

Question	**Answer options**		
Q15b: I would say that I write more	Yes	Uncertain	No
1. Understandable low-level tests with *Spectrum/Spock* than with JUnit?	A C		B
2. Maintainable low-level tests with *Spectrum/Spock* than with JUnit?	A C		B

Table 6.17: Developer perception towards *Spectrum/Spock*

Many of the made hypotheses get support from the results in this section. **H2:** *"Developers will find it easier to understand test cases"* seems to have more evidence as two out of three participants find test cases more understandable. **H3:** *"Developers will find it easier to maintain code"* has some results supporting it. Only *Spock* tests were perceived as easier to maintain than *JUnit* ones, but both *Spectrum* and *Spock* had developers that felt new tests with BDD-testing frameworks would act as easier to understand documentation for the production code. This living documentation is one the claims of BDD literature [41] and it seems to have merit. In total, H3 seems to hold true to an extent. **H4:** *"Developers will perceive working with low-level automated testing more enjoyable"* has two out of three developers agreeing with the statement. This could be interesting to study in a larger scale, as low enjoyment [5] and motivation [6] in unit testing was found problematic in earlier researches. With the results from this study, it can't be said with certainty that BDD-testing frameworks are the answer for enjoyment problem. Although there is more evidence to support H4 than not.

Summing up, participants A and C could be grouped closer together regarding developer perception changes with BDD-testing frameworks. This is somewhat interesting, as they use different frameworks. In total, the changes in their perception could be categorized to be on the positive side. Participant B could be categorized to perceive the change as indifferent or negative. Its interesting to see that participants A and B have same kind of experience as developers and they perceive low-level testing with *JUnit* the same, but the difference in perceiving the *Spectrum* changes was quite different.

6.2.2.2 Developer loyalty towards low-level testing

Question	**Answer options**										
	Not at all likely										*Extremely likely*
Q16: How likely are you to	0	1	2	3	4	5	6	7	8	9	10
1. Recommend low-level automated testing for colleague as a software development practice?									B		A C
2. Recommend testing framework JUnit for future Spring projects where you take part in existing project?								C		B	A
3. Take testing framework JUnit in use for future Spring projects where you have technical lead role in a new starting project?									C	B	A
	Not at all likely										*Extremely likely*
Q16': How likely are you to	0	1	2	3	4	5	6	7	8	9	10
1. Recommend low-level automated testing for colleague as a software development practice?									B		A C
2. Recommend testing framework *Spectrum/Spock* for future Spring projects where you take part in existing project?				B			A		C		
3. Take testing framework *Spectrum/Spock* in use for future Spring projects where you have technical lead role in a new starting project?				B			A		C		

Table 6.18: NPS questions related to *JUnit* and *Spectrum/Spock*

Developer loyalty towards low-level testing is measured with **NPS** survey questions. As the participant count is so low, it is not sensible to calculate the NPS directly. The information what can be used, is that for every sub-question in questions **Q16** and **Q16'** values from zero to six are called *detractors*, seven to eight *passives* and nine to ten *promoters* [73]. The interesting part here is to see, whether the participant has gained loyalty enough towards testing framework to be called a promoter.

Table 6.18 displays all the NPS survey questions with answers. Participants A and C can be categorized as promoters of low-level testing in general before and after the introduction of new BDD-testing framework. Participant B remains as a passive before and after the change.

Before the introduction of *Spectrum*, participant A is also a steady promoter of *JUnit* both as a technical lead in new project or taking part in existing project. After using *Spectrum* for two months, participant can be categorized as a detractor in loyalty towards *Spectrum* in *Spring Framework* context. This was further analyzed in interview where participant states that the found bugs in 1.1.0 version of *Spectrum* in IDE test output and build test output lower the score. He also states that *Spring Framework* support is not optimal yet and for these reasons he hesitates to fully get on board with *Spectrum*. The same effect is visible with participant B, as he was first a promoter of *JUnit*, but can later be categorized as a detractor for *Spectrum* NPS. In the interview he says that the lacking *Spring Framework* support is the main reason for not recommending *Spectrum*. Participant C remains as a passive both with *JUnit* and *Spock*, although he states in following interview that he would more probably use *Spock* in a new *Spring Framework* project than *JUnit*.

6.2.2.3 Change summary

Benefit	**Participants agreeing**
Enjoy writing tests more	A, C
Tests help to understand production code better	A, C
Promotes to write higher quality test code	A, C
More understandable tests	A, C
More maintainable tests	A, C
Maintaining is easier	C
Easier to write tests	C

Drawback	
More difficult to write tests	A, B
Would like to have more tool support	A, B
Would not recommend to use with Spring Framework	A, B
Slightly more difficult to maintain tests	A

Table 6.19: Developer perception changes of low-level testing

Table 6.19 summarizes results for **RQ3** - the perception changes towards low-level testing in Spring Framework context. The results can be categorized into two parts. First, participants A and C are close to each other in perceiving beneficial aspects in changing from *JUnit* to BDD-testing framework. Participant B could be categorized to be either neutral to changes or perceiving some drawbacks in BDD-testing compared to *JUnit*. Second, changing from *JUnit* to *Spectrum* is perceived to have more negative consequences than changing to *Spock*. The main findings for RQ3 are:

In low level testing, there is evidence to support that BDD-testing frameworks could rise enjoyment in testing. Results also show that BDD tests could help to understand implemented production code better while producing tests that are perceived as more understandable and maintanable. There is indication that BDD-testing frameworks can potentially promote to write higher quality test code than JUnit.

Spectrum was perceived to have more negative change consequenses than Spock. Whereas Spock tests were perceived as more easily writable and maintainable, this was not directly so with Spectrum. Spectrum was also seen as an early and unstable framework that needs more tool support.

6.3 Test code analysis

Test code analysis was done to answer **RQ3** with metrics defined in chapter 4 section 4.3.3.5. Test code analysis was done with master branches of projects A and B, **before** and **after** the introduction of new BDD-testing framework, to related low-level tests and their affected components. First, the unit testing level changes are inspected with projects A and B. After this, the low-level testing metrics and changes in them are demonstrated with both projects. Finally, the findings are summarized.

6.3.1 Automated unit testing level

At automated unit testing level, two metrics were used to study the changes in unit testing: average **count of test methods** per tested class methods and **code coverage**. Table 6.20 displays these values before the introduction of new BDD-testing framework and after. In **COTM**, before values were calculated for *JUnit* unit tests only and after values only for new unit tests done with the selected BDD-testing framework. In **CC**, before values are for the whole project at unit level with *JUnit*. After values are unit testing coverage for *JUnit* and new BDD-testing framework combined.

Metric	**Project A** JUnit	**Project A** Spectrum	**Project B** JUnit	**Project B** Spock
COTM	**1.44**	**5.63**	**3.49**	**25.5**
Sum of tested class methods	61	8	93	4
Sum of unit test methods	88	45	325 (294)*	102 (7)*
Instruction CC	**25%**	**24%**	**20%**	**19%**
Total number of instructions	31,425	44,427	49,895	53,211
Branch CC	**24%**	**26%**	**20%**	**21%**
Total number of branches	724	1,107	2,195	2,316

Table 6.20: Unit level testing metrics in projects and their change

* = Value in parenthesis is without calculating data-driven tests sum

RQ3: *How does behavior-driven testing frameworks change written low-level test cases and test code coverage compared to JUnit framework?* is first studied with data from unit testing metrics displayed in table 6.20. Figure 6.7 visualizes the change in **COTM**. Both projects display a **drastic change** in number of unit test methods per tested class method. With *JUnit*, project A had around one to two unit test methods per class method, but with *Spectrum* the average is from 5 to 6 test methods per class method. The change is around 4 times more granular use of test methods than before. The change in COTM in project A seems to correlate well with the original claims of Astels [33]: *xSpec family* testing helps in producing more granular tests without the tight one-to-one mapping of test methods to class methods.

At the starting point with *JUnit*, project B had first a fair amount of around 3 to 4 test methods per tested class method. With *Spock*, the average count of unit test methods per tested class method is at staggering 25 to 26 methods. In total there were 7 unit level feature methods with *Spock*, but all of them were data-driven tests. This resulted in 102 automatically generated separate test runs. As the number of study participants with *Spock* is low, this COTM-value might be an anomaly. Still, together with the later studied *DDTM*-metric, the analysis displays a significant growth in use DDT. The advertisement of *Spock*'s easy automatic test generation with DSL for data-driven testing [24] seems to have merit.

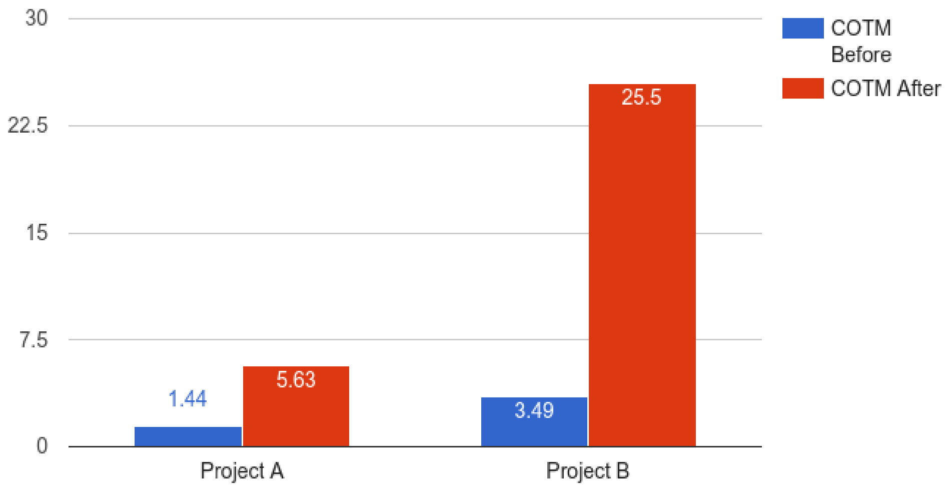


Figure 6.7: Average count of unit test methods per tested class method in projects

Both of studied BDD-testing frameworks showed an increase in test case granularity with test code analysis and participant practice survey. With the given study sample, **H1:** *"Developers will write more granular test cases"* holds true. This results in separately passing or failing test runs. In the original situation, less granular test cases could hold potentially multiple assertion errors at single test method run, but the first one halts the run of that test method. More granular test cases including more separate test methods with behavior describing naming should help in finding the erroneous situation faster. This could also potentially promote the test cases to act better as documentation for the tested production code. Documentation aspect of unit tests was identified in earlier as potential benefit in agile projects [6].

Code coverage didn't show almost any changes at unit level in either project. As participants earlier answered in the survey, none of them had code coverage as an optimizing target for low-level tests. This could explain the low coverage percentage seen in table 6.20. Although the test case granularity has risen a lot in both projects, it doesn't show up in test coverage one way or another. As such, it seems that CC is a poor metric to analyze changes in actual test code. This is further confirmed in next section with low-level test coverage examination.

6.3.2 Automated unit & integration testing levels

At automated low-level testing level there were 5 inspected metrics: **code coverage**, average **count of assertions** per test method, average **count of comments** per test method, average **test method name word count** and ratio of

data driven test methods to all low-level test methods. **CC** was measured as combined *JUnit* and new BDD framework coverage. All other metrics were measured first from existing low-level *JUnit* tests and after from new BDD-testing framework specifications. Table 6.21 dislays the values of these metrics in projects A and B, before and after introduction of BDD-testing framework.

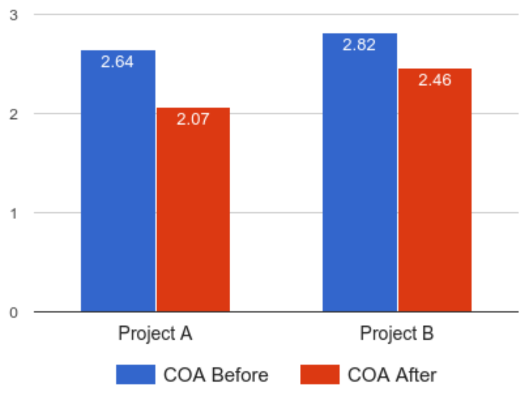
Metric	**Project A** JUnit	**Project A** Spectrum	**Project B** JUnit	**Project B** Spock
Instruction CC	**59%**	**58%**	**47%**	**46%**
Total number of instructions	31,425	44,427	49,895	53,211
Branch CC	**54%**	**54%**	**39%**	**40%**
Total number of branches	724	1,107	2,195	2,316
COA	**2.64**	**2.07**	**2.82**	**2.46**
Sum of assertions	493	174	1,311	32
Sum of test methods	187	84	465	13
COC	**1.17**	**0.08**	**0.04**	**0.07**
Sum of comments	218	7	20	1
Sum of test methods	187	84	465	13
TMNWC	**4.66**	**11.29**	**5.61**	**8.08**
Total words in test method names	872	948	2,607	105
Sum of test methods	187	84	465	13
DDTM	**0**	**0**	**0.02**	**0.54**
Sum of data driven test methods	0	0	8	7
Sum of test methods	187	84	465	13

Table 6.21: Low-level testing metrics in projects and their change

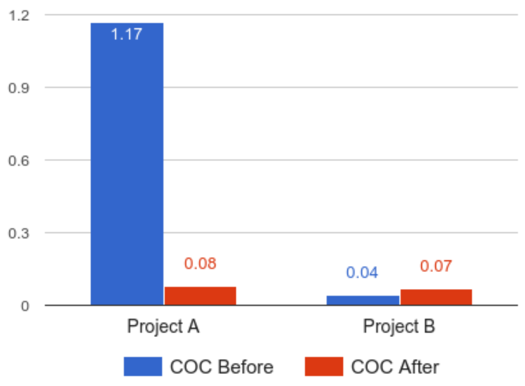
RQ3 can be further analyzed for low-level testing with metrics displayed in table 6.21. First studied metric is **CC**. With combined *JUnit* unit and integration tests, the instruction and branch coverage rises in both projects to around 50%. After the introduction of BDD-testing frameworks in projects, the combined CC for *JUnit* and new BDD tests remain virtually identical. It can be concluded, that although other studied metrics show quite significant changes, CC remains unchanged.

Figure 6.8 part a. displays the change in **COA** after the introduction of BDD-testing frameworks. Both of projects had closer to three than two assertions per test method with *JUnit*. With *Spectrum*, project A displayed a drop to approximately two assertions per test method. This is natural reaction to the more granular test cases - now there are more test methods that contain less assertions per method than before. The total count of assertions show an upward trend to earlier, as the total of 84 low-level test methods are a sum from far less test files (14 pcs.) than before (53 pcs. with *JUnit*).

Project B displayed smaller change in COA after the introduction of *Spock*. It is hard to say whether there would be a significant change in this with larger sample of test files to analyze. The structure of *Spock*, having assertions in *Then*-blocks, do not separate assertions to individual feature methods, instead one method can hold multiple blocks [45]. How this would affect COA in the long run, remains uncertain from these results.



((a)) Average count of assertions per test method



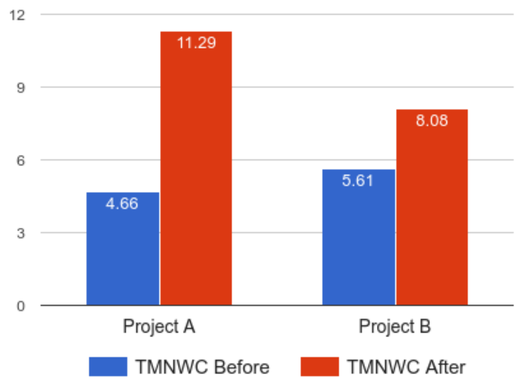
((b)) Average count of comments per test method

Figure 6.8: Test assertions and comments in projects

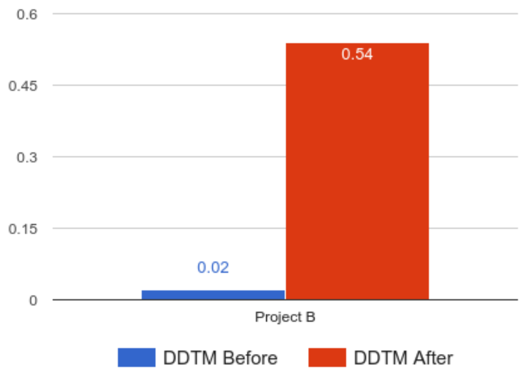
Figure 6.8 part b. shows the change in **COC** in projects. Commenting test cases and updating those comments was a practice that was found rarely done in earlier research [46]. Compared to this, project A showed to be an exception earlier with *JUnit*, as there was approximately at least one comment per test method. After the introduction of *Spectrum*, there was interesting change in test method commenting. The count of actual comments in tests dropped drastically to almost average zero comments per test method. When inspecting the change in **TMNWC** displayed in figure 6.9 part a, there seems to be a correlation with the drastic change in test commenting practices and the test naming practices. Info that was earlier inserted as pure comments, seem to be now a part of test naming with *describe* and *it* blocks. This also allows that info to be a part of test output, so for instance the failing test output could help the developer to grasp the test situation faster.

Project B with *Spock* showed virtually no change in commenting practices. There wasn't lot of commenting before and there still isn't commenting in place in tests with *Spock*. Interesting to see, is that also the possible textual descriptions of *Given-When-Then* blocks were missing. The change in TMNWC in project B was about 50% increase in test method name words, but as the block descriptions are optional, it is interesting to ponder how much more *Spock* actually promotes to write textual info into tests than JUnit. Unfortunately now, only one participant's *Spock* testing habbits were studied.

Earlier study [46] had identified developers preferring the tests to be self-documenting without commenting. Together with COC and TMNWC from both projects, there seems to be evidence to say that both BDD-testing frameworks promote this aspect better than *JUnit*. The text descriptions as part of test method naming seems to promote this self-documenting aspect of low-level tests. This is potentially more evidence for **H2:** *"Developers will find it easier to understand test cases"*, at least there exists less need for commenting of test methods and the test names hold more information through more words in them. Together with earlier survey and interview results, it can be concluded that H2 seems to hold true to an extent.



((a)) Average test method name word count



((b)) Ratio of data-driven tests to standard low-level test methods

Figure 6.9: Test method naming and DDT in projects

TMNWC changes are displayed in figure 6.9 part a. The relation between commenting practices and TMNWC was already discussed, but it's still good to highlight that in project A with *Spectrum*, there is on average almost 2.5 times words in test method names. The nested example groups and their code examples features in *xSpec family* testing seem to promote much better information on test method names than *JUnit*. In project B, *Spock* displayed a 50% increase in TMNWC. Tester habbits might be one of the reasons, but also the single line text description for test method naming could be a reason for a milder increase in words than observed *Spectrum*-tests.

DDTM metrics can be seen in figure 6.9 part b. Project A didn't have any DDT with *JUnit* at the start, nor did it have after the introduction of *Spectrum*. *Spectrum* and Java 8-lambdas allow automatic test method generation with usage through the language features, but the framework doesn't support this with a DSL. Earlier with *JUnit*, Project B had about 2% of all low-level test methods as data-driven. After the introduction of *Spock*, there was a drastic change: 54% of all feature methods were data-driven. *Spock* seems to promote DDT really well with its DSL and thus, at least maintaining of low-level tests with different test conditions should be easier.

6.3.3 Change summary

Metric	Project A	Project B
COTM	++	+++
CC	stagnant	stagnant
COA	-	-
COC	- - -	stagnant
TMNWC	+++	+
DDTM	stagnant	+++

Table 6.22: Test code analysis metrics change in projects

+ = Increase amount, - = Decrease amount

BDD-testing frameworks introduced changes to almost all of the studied testing metrics. *COTM* displayed a drastic change in both projects and especially with *Spock* and its data-driven testing. Both frameworks promote a lot more granular test cases than *JUnit*.

Code coverage remained virtually unchanged after the introduction of BDD-testing frameworks. In hindsight, it was fairly uninteresting metric to study this kind of testing transformation. It might have had different kind of results if actual BDD was practiced during the study.

COA had decreased in both projects. With project B, the amount of test files to study was fairly low, and thus, it is hard to draw real conclucions. Project A on the other hand displayed slightly more decrease in *COA*. This seems natural to the more granular use of test methods.

COC showed a significant change in project A test commenting practices. Earlier with *JUnit*, test commenting was an often used practice. After the introduction of *Spectrum*, this practice appeared to be substantially less used. Project B didn't have regular test commenting practices with *JUnit*. *Spock* didn't change the fact and test commenting remained a practice that was seldom used.

TMNWC changed in both projects. Project A had almost 2.5 times increased test method name word count with *Spectrum*. In project B, there was around

50% increase in *TMNWC*. Both BDD-testing families seem to promote more information on test names than *JUnit*. *Spectrum* and its internal structure seems to promote this even more than *Spock*. Together with the changes in test method naming and commenting, compared to *JUnit*, BDD-testing frameworks should produce more self-documenting test cases.

Data-driven testing was only in place in project B. Before with *JUnit*, DDT was used only a few times in total of low-level testing. With *Spock*, *DDTM*-metric increased tenfolds. Around half of the new tests with *Spock* were data-driven. It can be concluded that *Spock* promotes the use of data-driven testing much more than *JUnit*.

Table 6.22 summarises the results for **RQ3** - the changes in written low-level test and code coverage. The main findings are:

In low-level testing, BDD-testing frameworks produce more granular and self-documenting test cases than JUnit. Spock promotes the use of DDT more than JUnit.

Chapter 7

Conclusions

First in this chapter, the summary of case study and its results are presented. Second, the results are presented from the viewpoint of comparing studied implementation level BDD-testing frameworks. Third, possible directions for future work on the studied topic are discussed.

7.1 Summary

With today's trends in software development, agile practices are used more and more in producing software. What these practices usually have in common, is the incrementally built quality, where automated testing done by developers plays a crucial role. Traditionally developers use *xUnit family* testing frameworks for low-level automated testing. Although easy to get started, research has shown multiple possible problematic areas in the practice of unit testing with the said frameworks. This thesis studied alternative ways for low-level testing in the JVM context for testing Java-code. This was conducted as a case study in multi-project industry context. Idea was to see whether implementation level BDD-testing frameworks could help in problematic testing areas and how they change the practices and perception of low-level testing Java-code in general.

Practice changes in low-level testing with BDD-testing frameworks included developers writing *more granular* and *self-documenting* test cases. Especially the analyzed test code changes provided strong evidence for these changes. Other changes included developers feeling that it was *easier to identify separate test parts* than before with *JUnit*. There was also evidence to support *easier understanding of tests, more informative test output* and possibly *easier ways to remove repetition* from test code. Practice changes with BDD-testing frameworks were not only positive, as studied framework *Spectrum* displayed a somewhat *steep learning curve* compared to *JUnit*, with more time and effort needed in low-level testing.

Perception changes in low-level testing with BDD-testing frameworks displayed some *rise in enjoyment* in testing. There was also change in perception where two out of three practitioners perceived test done with BDD-testing frameworks more *understandable* and *maintanable* than before. Amongst the same participants, this was also accompanied together with the perception that BDD-testing frameworks *promote to write higher quality* test code than *JUnit.* One of the participants perceived changes in testing as neutral or slightly negative during the study period.

In conclusion, these results are quite promising as possible solutions to common problems found in unit testing, such as *low enjoyment* in testing together with *hard readability* and *maintainability* of tests. Future work section discusses in more detail how the use of implementation level BDD-testing frameworks in low-level testing could be studied further. Before that, next section summarizes the test changes as a comparison between used BDD-testing frameworks.

7.2 Comparison of BDD-testing frameworks

Studied aspect	**Spectrum**	**Spock**
Learning curve	Slow	Fast
More granular test cases	Yes	Yes
Easier to structure tests	Potentially	Yes
Easier to understand tests	Potentially	Yes
More enjoyable to test code	Potentially	Yes
More informative test output	Yes	Yes
More maintainable tests	Potentially	Yes
More self-documenting tests	Yes	Yes
Framework and tool support	Adequate	Good

Table 7.1: Summary of studied aspects of low-level testing with BDD frameworks

Table 7.1 summarizes the most important findings in this study from the viewpoint of the BDD-testing frameworks. There are two keypoints when comparing the alternatives: *Spock* offers more mature and easier to begin starting point for *Spring Framework* low-level testing than *Spectrum.*

Both produce more granular test cases than *JUnit*, where *Spock* achieves this with DDT and *Given-When-Then* blocks. *Spectrum* does this with more individually separated code examples describing behavior. *Spock* and its DDT should promote more easily maintainable tests, whereas *Spectrum* way of writing tests promotes individually passing or failing test conditions. *Spectrum* and its nested example groups with lifecycle hooks can be used for removing repetition from the code.

There is evidence to support that both can potentially allow more easily to structure and understand tests than *JUnit*. Although, the structure of *Spock* doesn't contain nesting, and as such might allow more easily to understand individual tests. With *Spectrum*, there exists a longer learning curve to 1earn how to structure and understand the nested structure properly. Especially the Java lambda-features used to structure *Spectrum* tests were found troublesome within the two months study time. Both BDD frameworks promote to write more self-documenting test cases than before. This is especially visible with *Spectrum*, where it can reduce the need for explicit commenting with test info embedded into test descriptive naming. This self-documenting aspect of tests is also visible in more information displayed in test output.

Summed up, these changes make both BDD frameworks good candidates to rise enjoyment in low-level testing compared to traditional *JUnit* testing. Although with *Spectrum*, the rise in enjoyment was not perceived unanimously. *Spock* could potentially rise enjoyment in low-level testing more quickly than *Spectrum*. *Spock*'s structure seems easier to learn when switching from *JUnit* and thus resulting more quickly in well structured tests.

7.3 Future work

Due to the limited scope of this study, together with constraints of master thesis, this case study has acted as preliminary research on the topic of comparing traditional unit testing framework *JUnit* against implementation level BDD-testing frameworks. This was conducted in an environment where BDD-testing frameworks were used as an alternative for *JUnit* in automated unit and integration testing, without the practice of BDD. There exists interesting possible research to continue on the topic of studying the use of implementation level BDD-testing frameworks for low-level testing.

Potential future work on the topic could include replicating this study on a larger scale on the JVM enviroment. This kind of study could also possibly be conducted in different environment, such as the *.NET* environment. With larger projects and number of study participants, it would be interesting to see whether the larger scale results follow the same trend as results gathered in this study. As many of the *xSpec family* drawbacks in this study was found out to come from the Java language features used in *Spectrum*, it could be one possible change in future studies to use alternative *xSpec family* framework. For example, framework from a dynamic programming language could be used, such as earlier demonstrated *RSpec* from JRuby.

Another future work possibility could be to further explore the studied aspects of this research in multiple isolated settings. This means using the same kind of survey that was used in this thesis to study for example long time Java developers using *JUnit*, Ruby developers using *RSpec* and Groovy developers using *Spock*. This kind of research could be done with a large number of participants and the results could be used to compare *xUnit family*, *xSpec family* and *Gherkin family* in low-level testing on a grand scale. It would offer insight on low-level testing practices and perception towards it, without the possible resistance to change that can surface in replicating exact same study as in this thesis. The test code analysis could also be done on a grand scale, for example mining public *GitHub* repositories and their *JUnit, RSpec* or *Spock* tests to calculate different metrics.

Bibliography

[1] L. Prechelt, H. Schmeisky, and F. Zieris, “Quality experience: a grounded theory of successful agile projects without dedicated testers,” in *Proceedings of the 38th International Conference on Software Engineering*, pp. 1017–1027, ACM, 2016.

[2] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development.* ” O’Reilly Media, Inc.”, 2004.

[3] D. North, “Introducing bdd.” `https://dannorth.net/introducing-bdd/`, 2006. [Online; accessed 24-March-2017].

[4] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesøy, B. Helmkamp, and D. North, *The RSpec Book: Behaviour-driven Development with RSpec, Cucumber, and Friends.* Pragmatic Bookshelf Series, Pragmatic Bookshelf, 2010.

[5] E. Daka and G. Fraser, “A survey on unit testing practices and problems,” in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pp. 201–211, IEEE, 2014.

[6] P. Runeson, “A survey of unit testing practices,” *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[7] Wikipedia, “Java virtual machine — wikipedia, the free encyclopedia.” `https://en.wikipedia.org/w/index.php?title=Java_virtual_machine&oldid=767900023`, 2017. [Online; accessed 21-March-2017].

[8] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification: Java SE 8 Edition.* Oracle America, 2015.

[9] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder, “Characteristics of dynamic jvm languages,” in *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pp. 11–20, ACM, 2013.

[10] F. Buckley, "A standard for software quality assurance plans," *Computer*, vol. 12, no. 8, 1978.

[11] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target [special issues section]," *IEEE Software*, vol. 13, pp. 12–21, Jan 1996.

[12] P. J. Denning, "Software quality," *Commun. ACM*, vol. 59, pp. 23–25, Aug 2016.

[13] L. McLeod and S. G. MacDonell, "Factors that affect software systems development project outcomes: A survey of research," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 24, 2011.

[14] N. Cerpa and J. M. Verner, "Why did your project fail?," *Communications of the ACM*, vol. 52, no. 12, pp. 130–134, 2009.

[15] R. Charette and J. Romero, "Lessons from a decade of it failures." `http://spectrum.ieee.org/static/lessons-from-a-decade-of-it-failures`, 2015. [Online; accessed 15-March-2017].

[16] M. Huo, J. Verner, L. Zhu, and M. A. Babar, "Software quality and agile methods," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pp. 520–525, IEEE, 2004.

[17] M. V. Mäntylä and J. Itkonen, "How are software defects found? the role of implicit defect detection, individual responsibility, documents, and knowledge," *Information and Software Technology*, vol. 56, no. 12, pp. 1597–1612, 2014.

[18] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 202–212, ACM, 2013.

[19] J. Itkonen, "Lecture: Building quality in modern software development," October 2016.

[20] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams.* Pearson Education, 2009.

[21] R. Osherove, *The Art of Unit Testing, Second Edition.* Manning Publications Company, 2013.

[22] J. A. Whittaker, "What is software testing? and why is it so hard?," *IEEE software*, vol. 17, no. 1, pp. 70–79, 2000.

[23] J. Langr, A. Hunt, and D. Thomas, *Pragmatic Unit Testing in Java 8 with JUnit.* Pragmatic Bookshelf, 2015.

[24] K. Kapelonis, *Java Testing with Spock.* Manning Publications Company, 2016.

[25] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Proceedings of the 7th International Workshop on Automation of Software Test*, pp. 36–42, IEEE Press, 2012.

[26] L. Williams, G. Kudrjavets, and N. Nagappan, "On the effectiveness of unit test automation at microsoft.," in *ISSRE*, pp. 81–89, 2009.

[27] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 571–579, IEEE, 2005.

[28] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[29] W. Bissi, A. G. S. S. Neto, and M. C. F. P. Emer, "The effects of test driven development on internal quality, external quality and productivity: A systematic review," *Information and Software Technology*, vol. 74, pp. 45–54, 2016.

[30] K. Beck, "Aim, fire [test-first coding]," *IEEE Software*, vol. 18, no. 5, pp. 87–89, 2001.

[31] S. Kollanus, "Test-driven development-still a promising approach?," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pp. 403–408, IEEE, 2010.

[32] M. F. Aniche and M. A. Gerosa, "Most common mistakes in test-driven development practice: Results from an online survey with developers," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 469–478, IEEE, 2010.

[33] D. Astels, "A new look at test-driven development," *http://blog. daveastels. com/files/BDD_Intro. pdf¿ Acessado em*, vol. 12, p. 2013, 2006.

[34] E. Amodeo, *Learning Behavior-driven Development with JavaScript.* Community Experience Distilled, Packt Publishing, 2015.

[35] S. Hammond and D. Umphress, "Test driven development: the state of the practice," in *Proceedings of the 50th Annual Southeast Regional Conference*, pp. 158–163, ACM, 2012.

[36] M. Gärtner, *ATDD by example: a practical guide to acceptance test-driven development.* Addison-Wesley, 2012.

[37] B. Haugset and T. Stalhane, "Automated acceptance testing as an agile requirements engineering practice," in *System Science (HICSS), 2012 45th Hawaii International Conference on*, pp. 5289–5298, IEEE, 2012.

[38] DevelopSense, "Blog: Acceptance tests: Letâs change the title, too." `http://www.developsense.com/blog/2010/08/acceptance-tests-lets-change-the-title-too/`, 2010. [Online; accessed 24-March-2017].

[39] J. H. Hayes, A. Dekhtyar, and D. S. Janzen, "Towards traceable test-driven development," in *Traceability in Emerging Forms of Software Engineering, 2009. TEFSE'09. ICSE Workshop on*, pp. 26–30, IEEE, 2009.

[40] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pp. 383–387, IEEE, 2011.

[41] J. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle.* Manning Publications Company, 2014.

[42] A. Okolnychyi and K. Fögen, "A study of tools for behavior-driven development," *Full-scale Software Engineering/Current Trends in Release Engineering*, p. 7, 2016.

[43] Wikipedia, "User story — wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=User_story&oldid=775339969`, 2017. [Online; accessed 2-May-2017].

[44] R. M. Lerner, "At the forge: Rspec," *Linux Journal*, vol. 2009, no. 186, 2009.

[45] P. Niederwieser, "Spock framework reference documentation." `http://spockframework.org/spock/docs/1.1-rc-3/all_in_one.html`, 2017. [Online; accessed 28-March-2017].

[46] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically documenting unit test cases," in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pp. 341–352, IEEE, 2016.

[47] L. Chantal and S. Regina, “Towards and empirical evaluation of behavior-driven development,” 2009.

[48] JUnit, “Junit - about.” `http://junit.org/junit4/`, 2017. [Online; accessed 22-March-2017].

[49] Github, “Roadmap - junit-team/junit5 wiki - github.” `https://github.com/junit-team/junit5/wiki/Roadmap`, 2017. [Online; accessed 22-March-2017].

[50] P. Software, “Spring framework.” `https://projects.spring.io/spring-framework/`, 2017. [Online; accessed 22-March-2017].

[51] Wikipedia, “Spring framework — wikipedia, the free encyclopedia.” `https://en.wikipedia.org/w/index.php?title=Spring_Framework&oldid=770621263`, 2017. [Online; accessed 22-March-2017].

[52] P. Software, “Spring integration testing.” `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/integration-testing.html`, 2017. [Online; accessed 22-March-2017].

[53] G. C. Archive, “The hamcrest tutorial.” `https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki`, 2017. [Online; accessed 04-April-2017].

[54] G. Inc, “Gradle build tool.” `https://gradle.org/`, 2017. [Online; accessed 30-March-2017].

[55] Artima, “Scalatest - getting started with funspec.” `http://www.scalatest.org/getting_started_with_fun_spec`, 2017. [Online; accessed 27-March-2017].

[56] RSpec, “Rspec documentation.” `http://rspec.info/documentation/`, 2017. [Online; accessed 28-March-2017].

[57] RSpec, “rspec-core.” `http://rspec.info/documentation/3.5/rspec-core/`, 2017. [Online; accessed 28-March-2017].

[58] Mockito, “Mockito framework site.” `http://site.mockito.org/`, 2017. [Online; accessed 30-March-2017].

[59] RSpec, “Explicit subject - subject - rspec core - rspec - relish.” `https://www.relishapp.com/rspec/rspec-core/v/3-5/docs/subject/explicit-subject`, 2017. [Online; accessed 28-March-2017].

[60] JetBrains, “Intellij idea the java ide.” `https://www.jetbrains.com/idea/`, 2017. [Online; accessed 30-March-2017].

[61] RubyGems, "What is a gem? - rubygems guides." `http://guides.rubygems.org/what-is-a-gem/`, 2017. [Online; accessed 30-March-2017].

[62] Github, "greghaskins/spectrum: A bdd-style test runner for java 8. inspired by jasmine, rspec, and cucumber." `https://github.com/greghaskins/spectrum/`, 2017. [Online; accessed 27-March-2017].

[63] Github, "greghaskins/spectrum: A bdd-style test runner for java 8. inspired by jasmine, rspec, and cucumber." `https://github.com/greghaskins/spectrum/tree/1.0.2`, 2017. [Online; accessed 27-March-2017].

[64] Github, "Gherkin." `https://github.com/cucumber/cucumber/wiki/Gherkin`, 2017. [Online; accessed 28-March-2017].

[65] Github, "jimweirich/rspec-given: Given/when/then keywords for rspec specifications." `https://github.com/jimweirich/rspec-given`, 2017. [Online; accessed 28-March-2017].

[66] Artima, "Scalatest - getting started with featurespec." `http://www.scalatest.org/getting_started_with_feature_spec`, 2017. [Online; accessed 28-March-2017].

[67] Github, "Pease." `http://pease.github.io/`, 2017. [Online; accessed 28-March-2017].

[68] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[69] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons, 2012.

[70] D. Cohen and B. Crabtree, "Qualitative research guidelines project." `http://www.sswm.info/sites/default/files/reference_attachments/COHEN%202006%20Semistructured%20Interview.pdf`, 2006. [Online; accessed 06-April-2017].

[71] LimeSurvey, "Limesurvey: the online survey tool - open source suryeys." `https://www.limesurvey.org/`, 2017. [Online; accessed 06-April-2017].

[72] R. A. Cummins and E. Gullone, "Why we should not use 5-point likert scales: The case for subjective quality of life measurement," in *Proceedings, second international conference on quality of life in cities*, pp. 74–93, 2000.

[73] F. F. Reichheld, "The one number you need to grow," *Harvard business review*, vol. 81, no. 12, pp. 46–55, 2003.

[74] EclEmma, "Eclemma jacoco java code coverage library." `http://www.eclemma.org/jacoco/`, 2017. [Online; accessed 07-April-2017].

[75] M. G. . C. KG and Contributors, "Jacoco coverage counter." `http://www.jacoco.org/jacoco/trunk/doc/counters.html`, 2017. [Online; accessed 07-April-2017].

[76] Wikipedia, "Camel case — wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Camel_case&oldid=773178217`, 2017. [Online; accessed 07-April-2017].

[77] T. A. S. Foundatation, "Maven 2013; introduction." `https://maven.apache.org/what-is-maven.html`, 2017. [Online; accessed 18-April-2017].

[78] Github, "Parameterized tests - junit-team/junit4 wiki." `https://github.com/junit-team/junit4/wiki/parameterized-tests`, 2017. [Online; accessed 12-April-2017].

Appendix A

Interview questions

A.1 Interview for demographic purposes

1. How long have you worked as a software developer?
2. How long have you worked as a Java developer?
    (a) How long in the Spring Framework context?
3. What is the project you are working on now?
    (a) What is the context of project?
        i. Industry branch or government related?
        ii. What is the nature of software development?
    (b) Does the project use agile or traditional process?
    (c) What is the size of the project?
    (d) What is the size of development team?
    (e) How would you describe the project's software architecture?
    (f) What practices are in the used quality assurance process?
    (g) What is the used automated testing framework for
        i. Unit-level?
        ii. Integration-level?
4. How much experience do you have with automated
    (a) Unit testing and with what frameworks?
    (b) Integration testing and with what frameworks?
5. How much experience do you have with automated behavior-driven development testing frameworks before this experiment?

Appendix B

Surveys

This appendix explains in detail the two surveys used in this case study: their questions and relationship to related research surveys & found problems. First, the survey regarding JUnit is examined, followed by the inspection of BDD-testing framework survey.

B.1 Survey regarding JUnit in automated low-level testing

Survey regarding JUnit contains base questions from multiple previous research studies [5, 26, 46]. They were used as base to see how the participants in this study are positioned related to other studied practitioners. Related research in chapter 2 section 2.7 highlighted findings from previous studies. This survey is aimed to study about the participants practices and perception related to common problems and other findings found in earlier research. Table B.1 displays how JUnit survey questions are related to previous studies and to which research questions they will later help answering:

Related research questions	**Survey questions**	**Related research findings**
RQ1	Q3	Developers are mainly trying to find realistic scenarios on what to test [5]
RQ1	Q11, Q12	Developers finding isolating of unit under test hard [5]
RQ2	Q14	Only half of the survey respondents enjoy writing unit tests [5, 6]
RQ1, RQ2	Q2, Q7, Q14, Q15	Maintaining unit tests was found hard [5, 6]
RQ1	Q4, Q5, Q6	For 60.38% of developers, understanding unit tests is at least moderately difficult [46]
RQ1	Q8, Q9	Developers find updated documentation and comments in test cases useful, but writing comments to unit tests is rarely or never done [46]
RQ2	Q14, Q15	Majority of developers find unit tests helpful in producing higher quality code [26]
RQ2	Q14	Majority of developers find unit tests helpful in understanding other peoples code [26]
RQ1	Q1, Q10	
RQ2	Q16	

Table B.1: JUnit survey questions related to research questions and earlier studies

Tables B.2, B.3 and B.4 display all the questions in the JUnit survey. They contain multiple copied questions from other studies. All the copied questions originally had the word *'unit'* in them instead of *'low-level'*. This change was made to accompany both unit and integration testing into the survey of this study.

First copied questions are questions **Q1:** *How do you spend your software development time?* and **Q3:** *How important are the following aspects for you when you write new low-level tests?* [5]. These questions were copied to study as a base how the development time is used and what aspects are priortized in low-level testing. Q3 Sub-questions 1 and 3-8 are originals. Q3 Sub-question 2 is added to this survey to study the initial attitude towards describing behavior in tests.

Third copied question is **Q4:** *How difficult is it for you usually to understand a low-level test?* [46]. This question was added to survey to get a general concensus about understandability of the low-level testing within participants. Fourth and fifth copied questions are **Q8:** *How often do you add/write documentation comments to low-level test cases?* and **Q9:** *When you make changes to low-level tests, how often do you comment the changes (or update existing comments)?* [46]. These questions were copied to study the low-level testing documentation practices.

Sixth and seven copied questions are **Q14 & Q15:** *Please indicate your level of agreement with the following statements.* Q14 and its sub-questions 1-5 are copied from *"A survey on unit testing practices and problems"* by Daka and Fraser [5]. Sub-questions 6-8 are original questions defined in this thesis. Q15 sub-question 1 is copied from a unit test practitioner survey done at Microsoft [26] and sub-question 2 is copied from unit test documentation practices survey by Li et al. [46]. Questions Q14 & Q15 were copied to gather developer perception towards low-level testing.

Question	**Answer options**					
Q1: How do you spend your software development time (in percentages)	0-100%					
1. Writing new code						
2. Writing new tests						
3. Debugging/fixing						
4. Refactoring						
5. Other						
Q2: How do you spend your low-level automated testing time	minutes / 0-100%					
1. How much approximately you use time per test case (minutes)?						
2. How much of your initial effort goes to thinking about test case content without implementation (percentage)?						
3. How much of your initial effort goes to initial test case structuring and implementation (percentage)?						
4. How much of your overall testing effort goes to refactoring test code (percentage)?						
Q3: How important are the following aspects for you when you write new low-level tests?	Not at all	Low importance	Slightly important	Neutral	Moderately important	Very important	Extremely important
1. Code coverage							
2. Capturing all behavior of unit/feature with tests or assertions							
3. Execution speed							
4. Robustness against code changes (i.e., test does not break easily)							
5. How realistic the test scenario is							
6. How easily faults can be localised/debugged if the test fails							
7. How easily the test can be updated when the underlying code changes							
8. Sensitivity against code changes (i.e., test should detect even small code changes)							
Q4: How difficult is it for you to understand a low-level test?	Very easy	Easy	Moderate	Hard	Very hard		
Q5: In low-level testing, how difficult is it for you to	Very easy	Easy	Slightly easy	Moderate	Slightly hard	Hard	Very hard
1. Structure and write information to context of test?							
2. Structure and write information to stimulus of test?							
3. Structure and write information to assertions of test?							
4. Read test case structure for information about context of test?							
5. Read test case structure for information about stimulus of test?							
6. Read test case structure for information about assertions of test?							
Q6: How informative you usually find the test output?	Not at all	Hardly informative	Slightly informative	Somewhat informative	Moderately informative	Very informative	Extremely informative
Q7: How much are the following repetition reducing techniques used in your low-level testing?	Never	Very rarely	Rarely	Occasionally	Frequently	Very frequently	Always
1. Extract method (custom helper methods)							
2. Lifecycle hooks Before/After -class							
3. Lifecycle hooks Before/After (each)							
4. Automatic test generation via test method parametrization							
5. Common test initializer class inheritance							
Q8: How often do you add/write documentation comments to low-level test cases?	Never	Rarely	Moderate	Hard	Very hard		
Q9: When you make changes to low-level tests, how often do you comment the changes (or update existing comments)?	Never	Rarely	Moderate	Hard	Very hard		
Q10: In unit testing, how many	0	1	2-3	4-5	6-7	8-9	10 or more
1. Test methods do you usually write per class method?							
2. Assertions do you usually write per test method?							
Q11: In unit testing, what mocking library do you normally use?	Mockito	jMock	Powermock	Easymock	Other		
Q12: In unit testing, how difficult you find it to	Very easy	Easy	Slightly easy	Moderate	Slightly hard	Hard	Very hard
1. Mock objects?							
2. Stub method calls?							
3. Verify mock object actions?							
Q13: When do you add automated unit tests for developed code?	Before implementation	During implementation	After implementation				

Table B.2: JUnit developer low-level testing practice questions

Question	Answer options						
Q14: Please indicate your level of agreement with the following statements	Strongly disagree	Disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Agree	Strongly agree
1. Writing low-level tests is difficult 2. I enjoy writing low-level tests 3. I would like to have more tool support when writing low-level tests 4. I would like to have more low-level tests 5. Maintaining low-level tests is difficult 6. I think my low-level tests will help other developers to understand the implemented unit/feature better 7. Low-level automated testing helps me find defects in the code before other quality assurance phases 8. JUnit promotes me to write high quality test code							
Q15: Please indicate your level of agreement with the following statements	Strongly disagree	Disagree	Neutral	Agree	Strongly agree		
1. Overall, low-level tests help me produce higher quality code 2. Maintaining good low-level test cases and their documentations is important to the quality of a system							

Table B.3: Likert scale questions of developer perception towards JUnit

Question	Answer options										
Q16: How likely are you to	0	1	2	3	4	5	6	7	8	9	10
1. Recommend low-level automated testing for colleague as a software development practice? 2. Recommend testing framework JUnit for future Spring projects where you take part in existing project? 3. Take testing framework JUnit in use for future Spring projects where you have technical lead role in a new starting project?											

Table B.4: NPS questions of developer loyalty towards low-level automated testing with JUnit

B.2 Survey regarding BDD-testing framework in automated low-level testing

Survey regarding BDD-testing framework was aimed for the projects A and B explained in chapter 5. This means that for project A, the survey was aimed at to study the differences between JUnit and *xSpec family* testing done with ***Spectrum***. For project B, survey was targeted to study the change from JUnit to *Gherkin family* ***Spock*** testing framework. BDD-testing framework survey questions, together with base JUnit survey questions, are used to answer to researh questions RQ1 and RQ2. Table B.5 demonstrates how BDD-testing framework survey questions are related to previous research findings and research questions in this thesis:

Related research questions	**Survey questions**	**Related research findings**
RQ1	Q3’	Developers are mainly trying to find realistic scenarios on what to test [5]
RQ1	Q11’, Q12’	Developers finding isolating of unit under test hard [5]
RQ2	*Q14’	Only half of the survey respondents enjoy writing unit tests [5, 6]
RQ1, RQ2	Q2’, Q7’, *Q14’, Q15	Maintaining unit tests was found hard [5, 6]
RQ1	Q4’, Q5’, Q6’, Q15	For 60.38% of developers, understanding unit tests is at least moderately difficult [46]
RQ1	Q8’, Q9’	Developers find updated documentation and comments in test cases useful, but writing comments to unit tests is rarely or never done [46]
RQ2	*Q14’	Majority of developers find unit tests helpful in producing higher quality code [26]
RQ2	*Q14’, Q15	Majority of developers find unit tests helpful in understanding other peoples code [26]
RQ1	Q1’, Q10’	
RQ2	Q16’	

Table B.5: JUnit survey questions related to research questions and earlier studies

* = Q14’ contains direct comparison to JUnit survey questions Q14 and Q15

Questions marked with ’-character are direct comparison questions to question with same number in JUnit survey displayed in section B.1. For example original question **Q2** in JUnit survey: *”How do you spend your low-level automated testing time?”* has its direct comparison question **Q2’**: *”Compared to JUnit, How do you* *spend your low-level automated testing time?"*. Although there were individual surveys used for each *Spock* and *Spectrum*, their questions are both displayed in same tables. The only change is in the words *Spectrum/Spock*, which are marked in questions. Full questions used in BDD surveys are displayed in tables B.6, B.7, B.8 and B.9.

Question	Answer options						
Q1': Compared to JUnit, How do you spend your software development time? 1. Writing new code 2. Writing new tests 3. Debugging/fixing 4. Refactoring 5. Other	A lot less time	Less time	Slightly less time	The same amount	Slightly more time	More time	A lot more time
Q2': Compared to JUnit, How do you spend your low-level automated testing time? 1. Do you use more or less time per test case? 2. Do you use more or less of initial effort thinking about test case content? (no implementation) 3. Do you use more or less of initial effort to test case structuring and implementation? 4. Do you use more or less of overall testing effort to refactoring test code?	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
Q3': Compared to JUnit, How important are the following aspects for you when you write new low-level tests? 1. Code coverage 2. Capturing all behavior of unit/feature with tests or assertions 3. Execution speed 4. Robustness against code changes (i.e., test does not break easily) 5. How realistic the test scenario is 6. How easily faults can be localised/debugged if the test fails 7. How easily the test can be updated when the underlying code changes 8. Sensitivity against code changes (i.e., test should detect even small code changes)	A lot less important	Less important	Slightly less important	As important as before	Slightly more important	More important	A lot more important
Q4': Compared to JUnit, how difficult is it for you to understand a low-level test?	A lot less difficult	Less difficult	Slightly less difficult	As difficult as before	Slightly more difficult	More difficult	A lot more difficult
Q5': Compared to JUnit in low-level testing, how difficult is it for you to. 1. Structure and write information to context of test? 2. Structure and write information to stimulus of test? 3. Structure and write information to assertions of test? 4. Read test case structure for information about context of test? 5. Read test case structure for information about stimulus of test? 6. Read test case structure for information about assertions of test?	A lot less difficult	Less difficult	Slightly less difficult	As difficult as before	Slightly more difficult	More difficult	A lot more difficult
Q6': Compared to JUnit, how informative you usually find the test output?	A lot less informative	Less informative	Slightly less informative	As informative as before	Slightly more informative	More informative	A lot more informative
Q7': Compared to JUnit, how much are the following repetition reducing techniques used in your low-level testing? 1. Extract method (custom helper methods) 2. Lifecycle hooks Before/After -class 3. Lifecycle hooks Before/After (each) 4. Automatic test generation via test method parametrization 5. Common test initializer class inheritance	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
Q8': Compared to JUnit, how often do you add/write documentation comments to low-level test cases?	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
Q9': Compared to JUnit when you make changes to low-level tests, how often do you comment the changes (or update existing comments)?	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
Q10': Compared to JUnit in unit testing 1. Do you write more or less test methods per class method? 2. Do you write more or less assertions per test method?	A lot less	Less	Slightly less	The same amount	Slightly more	More	A lot more
Q11': In unit testing with Spectrum/Spock, what mocking library do you normally use?	Mockito	jMock	Powermock	Easymock	Other	*Spock's internal mocking*	
Q12': Compered to JUnit in unit testing, how difficult you find it to 1. Mock objects? 2. Stub method calls? 3. Verify mock object actions?	A lot easier	Easier	Slightly easier	As difficult as before	Slightly harder	Harder	A lot harder
Q13': When do you add automated unit tests for developed code?	Before implementation	During implementation	After implementation				

Table B.6: *Spectrum/Spock* developer low-level testing practice questions

Question	**Answer options**						
Q14': Please indicate your level of agreement with the following statements	Strongly disagree	Disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Agree	Strongly agree
1. Writing low-level tests with *Spectrum/Spock* is more difficult than with JUnit							
2. I enjoy writing low-level tests with *Spectrum/Spock* more than I do with JUnit							
3. I would like to have more tool support for *Spectrum/Spock* when writing low-level tests							
4. I would like to have more low-level tests with *Spectrum/Spock*							
5. Maintaining low-level tests with *Spectrum/Spock* is more difficult than with JUnit							
6. I think my low-level tests with *Spectrum/Spock* will help other developers to understand the implemented unit/feature better than earlier tests with JUnit							
7. Low-level automated testing with *Spectrum/Spock* helps me find defects in the code before other quality assurance phases better than earlier tests with JUnit							
8. *Spectrum/Spock* promotes me to write higher quality test code than with JUnit							
9. Overall, low-level tests with *Spectrum/Spock* help me produce higher quality code than with JUnit							
10. Maintaining good low-level test cases and their documentation with *Spectrum/Spock* is more important for system quality than maintaining JUnit test cases							

Table B.7: Likert scale questions of developer perception towards *Spectrum/Spock*

Question	**Answer options**		
Q15: I would say that I write more	Yes	Uncertain	No
1. Understandable low-level tests with *Spectrum/Spock* than with JUnit?			
2. Maintainable low-level tests with *Spectrum/Spock* than with JUnit?			

Table B.8: Questions of developer perception towards *Spectrum/Spock*

Question	**Answer options**										
Q16': How likely are you to	0	1	2	3	4	5	6	7	8	9	10
1. Recommend low-level automated testing for colleague as a software development practice?											
2. Recommend testing framework *Spectrum/Spock* for future Spring projects where you take part in existing project?											
3. Take testing framework *Spectrum/Spock* in use for future Spring projects where you have technical lead role in a new starting project?											

Table B.9: NPS questions of developer loyalty towards low-level automated testing with *Spectrum/Spock*

Appendix C

Gradle build configurations

```
//integration test compile task
configurations {
    integrationTestCompile.extendsFrom testCompile
    integrationTestRuntime.extendsFrom testRuntime
}

dependencies {
    testCompile 'junit:junit:' + junitVersion
    testCompile 'org.hamcrest:hamcrest-all:' + hamcrestVersion //hamcrest-matchers
    testCompile 'org.mockito:mockito-core:' + mockitoVersion //mockito for test isolation
    testCompile 'io.rest-assured:rest-assured:' + restAVersion //integration-testing REST
    //spring domain extension
    integrationTestCompile 'org.springframework.boot:spring-boot-starter-test'
}

//add integration test sources
sourceSets {
    integrationTest {
        java {
            compileClasspath += main.output + test.output
            runtimeClasspath += main.output + test.output
            srcDir file('src/integration-test/java')
        }
        resources.srcDir file('src/integration-test/resources')
    }
}

//add integration test task
task integrationTest(type: Test) {
    testClassesDir = sourceSets.integrationTest.output.classesDir
    classpath = sourceSets.integrationTest.runtimeClasspath
    outputs.upToDateWhen { false }
}

check.dependsOn test // gradle build runs unit tests
check.dependsOn integrationTest // gradle build runs integration tests
integrationTest.mustRunAfter test // integration tests are run after unit tests
```

Figure C.1: Relevant parts of JUnit Gradle build configuration

```
//integration test compile task
configurations {
    integrationTestCompile.extendsFrom testCompile
    integrationTestRuntime.extendsFrom testRuntime
}

dependencies {
    testCompile 'junit:junit:' + junitVersion
    testCompile 'org.hamcrest:hamcrest-all:' + hamcrestVersion //hamcrest-matchers
    testCompile 'org.mockito:mockito-core:' + mockitoVersion //mockito for test isolation
    testCompile 'io.rest-assured:rest-assured:' + restAVersion //integration-testing REST
    testCompile 'com.greghaskins:spectrum:' + spectrumVersion //spectrum junit-runner
    //spring domain extension
    integrationTestCompile 'org.springframework.boot:spring-boot-starter-test'
}

//add integration test sources
sourceSets {
    integrationTest {
        java {
            compileClasspath += main.output + test.output
            runtimeClasspath += main.output + test.output
            srcDir file('src/integration-test/java')
        }
        resources.srcDir file('src/integration-test/resources')
    }
}

//add integration test task
task integrationTest(type: Test) {
    testClassesDir = sourceSets.integrationTest.output.classesDir
    classpath = sourceSets.integrationTest.runtimeClasspath
    include '**/*Spec.java'
    outputs.upToDateWhen { false }
}

check.dependsOn test // gradle build runs unit tests
check.dependsOn integrationTest // gradle build runs integration tests
integrationTest.mustRunAfter test // integration tests are run after unit tests
```

Figure C.2: Relevant parts of Spectrum Gradle build configuration

```
//integration test compile task
configurations {
    integrationTestCompile.extendsFrom testCompile
    integrationTestRuntime.extendsFrom testRuntime
}

dependencies {
    //groovy rest-client
    testCompile 'org.codehaus.groovy.modules.http-builder:http-builder:0.6'
    testCompile 'org.codehaus.groovy:groovy-all:2.4.4' //groovy
    testCompile 'org.spockframework:spock-core:1.1-groovy-2.4-rc-3' //spock
    testCompile 'cglib:cglib-nodep:3.2.4' // stubbing java classes
    //bdd html reports
    testCompile ('com.athaydes:spock-reports:1.2.13') {
        transitive = false // this avoids affecting your version of Groovy/Spock
    }
    //mockito InjectMocks type mocking (not recommended)
    testCompile 'com.blogspot.toomuchcoding:spock-subjects-collaborators-extension:1.2.1'
    //spring extension for spock
    integrationTestCompile 'org.spockframework:spock-spring:1.1-groovy-2.4-rc-3'
}

//add integration test sources for Spock
sourceSets {
    integrationTest {
        groovy {
            compileClasspath += main.output + test.output
            runtimeClasspath += main.output + test.output
            srcDir file('src/integration-test/groovy')
        }
        resources.srcDir file('src/integration-test/resources')
    }
}

//add integration test task
task integrationTest(type: Test) {
    testClassesDir = sourceSets.integrationTest.output.classesDir
    classpath = sourceSets.integrationTest.runtimeClasspath
    include '**/*Spec.*'
    outputs.upToDateWhen { false }
}

check.dependsOn test // gradle build runs unit tests
check.dependsOn integrationTest // gradle build runs integration tests
integrationTest.mustRunAfter test // integration tests are run after unit tests
```

Figure C.3: Relevant parts of Spock Gradle build configuration

```
configurations { rspec }
dependencies {
    //ruby
    rspec 'org.jruby:jruby-complete:' + jRubyVersion
}

//install bundler for handling ruby gems
task(bundler, type: JavaExec) {
    main = 'org.jruby.Main'
    classpath = configurations.rspec
    args = ['-S', 'gem', 'install', 'bundler']
    environment['GEM_PATH'] = file('build/gems').path
    environment['GEM_HOME'] = file('build/gems').path
}

//install gems from gemfile with bundler
task(gems, dependsOn: ["bundler"], type: JavaExec) {
    main = 'org.jruby.Main'
    classpath = configurations.rspec
    args = ['-S', 'bundle', 'install']
    environment['GEM_PATH'] = file('build/gems').path
    environment['GEM_HOME'] = file('build/gems').path
}

//run all or single unit spec(s)
task(spec, dependsOn: ["classes"], type: JavaExec) {
    main = 'org.jruby.Main'
    classpath = sourceSets.test.runtimeClasspath + configurations.rspec
    if ( project.hasProperty("file") ) {
        args = ['-S', 'build/gems/bin/rspec', 'src/spec/unit/'+file]
    } else {
        args = ['-S', 'build/gems/bin/rspec', 'src/spec/unit']
    }
    environment['GEM_HOME'] = file('build/gems').path
    environment['GEM_PATH'] = file('build/gems').path
    environment['ENV'] = "test"
}

//run all or single integration spec(s)
task(integrationSpec, dependsOn: ["classes"], type: JavaExec) {
    main = 'org.jruby.Main'
    classpath = sourceSets.test.runtimeClasspath + configurations.rspec
    if ( project.hasProperty("file") ) {
        args = ['-S', 'build/gems/bin/rspec', 'src/spec/api/'+file]
    } else {
        args = ['-S', 'build/gems/bin/rspec', 'src/spec/api']
    }
    environment['GEM_HOME'] = file('build/gems').path
    environment['GEM_PATH'] = file('build/gems').path
    environment['ENV'] = "integration-test"
}

check.dependsOn spec // gradle build runs unit specs
check.dependsOn integrationSpec // gradle build runs integration specs
integrationSpec.mustRunAfter spec // integration specs are run after unit specs
```

Figure C.4: Relevant parts of RSpec Gradle build configuration