

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 frame- works vs JUnit

Master's Thesis
Espoo, June 18, 2011

DRAFT! — April 12, 2017 — DRAFT!

Supervisor: Professor Antti Ylä-Jääski, Aalto University
Professor Pekka Perustieteilijä, University of Helsinki
Advisor: Olli Ohjaaja M.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:	June 18, 2011	Pages:	ix + 81
Major:	Data Communication Software	Code:	T-110
Supervisor:	Professor Antti Ylä-Jääski Professor Pekka Perustieteilijä		
Advisor:	Olli Ohjaaja M.Sc. (Tech.)		
<p>A dissertation or thesis is a document submitted in support of candidature for a degree or professional qualification presenting the author's research and findings. In some countries/universities, the word thesis or a cognate is used as part of a bachelor's or master's course, while dissertation is normally applied to a doctorate, whilst, in others, the reverse is true.</p> <p>!FIXME Abstract text goes here (and this is an example how to use fixme). FIXME! Fixme is a command that helps you identify parts of your thesis that still require some work. When compiled in the custom <code>mydraft</code> mode, text parts tagged with <code>fixmes</code> are shown in bold and with <code>fixme</code> tags around them. When compiled in normal mode, the <code>fixme</code>-tagged text is shown normally (without special formatting). The draft mode also causes the "Draft" text to appear on the front page, alongside with the document compilation date. The custom <code>mydraft</code> mode is selected by the <code>mydraft</code> option given for the package <code>aalto-thesis</code>, near the top of the <code>thesis-example.tex</code> file.</p> <p>The thesis example file (<code>thesis-example.tex</code>), all the chapter content files (<code>1introduction.tex</code> and so on), and the Aalto style file (<code>aalto-thesis.sty</code>) are commented with explanations on how the Aalto thesis works. The files also contain some examples on how to customize various details of the thesis layout, and of course the example text works as an example in itself. Please read the comments and the example text; that should get you well on your way!</p>			
Keywords:	ocean, sea, marine, ocean mammal, marine mammal, whales, cetaceans, dolphins, porpoises		
Language:	English		

Aalto-yliopisto
 Perustieteiden korkeakoulu
 Tietotekniikan koulutusohjelma

DIPLOMITYÖN
 TIIVISTELMÄ

Tekijä:	Antti Ahonen		
Työn nimi:	Ohjelmistoprosessit määntelle: Uusi organisaatio, uudet pyörät		
Päiväys:	18. kesäkuuta 2011	Sivumäärä:	ix + 81
Pääaine:	Tietoliikenneohjelmistot	Koodi:	T-110
Valvoja:	Professori Antti Ylä-Jääski Professori Pekka Perustieteilijä		
Ohjaaja:	Diplomi-insinööri Olli Ohjaaja		
<p>Kivi on materiaali, joka muodostuu mineraaleista ja luokitellaan mineraalisältönsä mukaan. Kivet luokitellaan yleensä ne muodostaneiden prosessien mukaan magmakiviin, sedimenttikiviin ja metamorfisiin kiviin. Magmakivet ovat muodostuneet kiteytyneestä magmasta, sedimenttikivet vanhempien kivilajien rapautuessa ja muodostaessa iskostuneita yhdisteitä, metamorfiset kivet taas kun magma- ja sedimenttikivet joutuvat syvällä maan kuoressa lämpötilan ja kovan paineen alaiseksi.</p> <p>Kivi on epäorgaaninen eli elottoman luonnon aine, mikä tarkoittaa ettei se sisällä hiiltä tai muita elollisen orgaanisen luonnon aineita. Niinpä kivistä tehdyt esineet säilyvät maaperässä tuhansien vuosien ajan mätänemättä. Kun orgaaninen materiaali jättää jälkensä kiveen, tulos tunnetaan nimellä fossiili.</p> <p>Suomen peruskallio on suurimmaksi osaksi graniittia, gneissiä ja Kaakkois-Suomessa rapakiveä.</p> <p>Kiveä käytetään teollisuudessa moniin eri tarkoituksiin, kuten keittiötasoihin. Kivi on materiaalina kalliimpaa mutta kestävämpää kuin esimerkiksi puu.</p>			
Asiasanat:	AEL, aineistot, aitta, akustiikka, Alankomaat, aluerakentaminen, Anttolanhovi, Arcada, ArchiCad, arkki		
Kieli:	Englanti		

Aalto-universitetet
Högskolan för teknikvetenskaper
Examensprogram för datateknik

**SAMMANDRAG AV
DIPLOMARBETET**

Utfört av:	Antti Ahonen		
Arbetets namn:	Den stora stygga vargen: Lilla Vargens universum		
Datum:	Den 18 Juni 2011	Sidantal:	ix + 81
Huvudämne:	Datakommunikationsprogram	Kod:	T-110
Övervakare:	Professor Antti Ylä-Jääski Professor Pekka Perustieteilijä		
Handledare:	Diplomingenjör Olli Ohjaaja		
<p>Lilla Vargens universum är det tredje fiktiva universumet inom huvudfåran av de tecknade disneyserierna - de övriga två är Kalle Ankas och Musse Piggs universum. Figurerna runt Lilla Vargen kommer huvudsakligen från tre källor — dels persongalleriet i kortfilmen Tre små grisar från 1933 och dess uppföljare, dels långfilmen Sången om Södern från 1946, och dels från episoden “Bongo” i långfilmen Pank och fågelfri från 1947. Framför allt de två första har sedermera även kommit att leva vidare, utvidgas och införlivas i varandra genom tecknade serier, främst sådana producerade av Western Publishing för amerikanska Disneytidningar under åren 1945–1984.</p> <p>Världen runt Lilla Vargen är, i jämförelse med den runt Kalle Anka eller Musse Pigg, inte helt enhetlig, vilket bland annat märks i Bror Björns skiftande personlighet. Den har även varit betydligt mer öppen för influenser från andra Disneyvärldar, inte minst de tecknade långfilmerna. Ytterligare en skillnad är att varelserna i vargserierna förefaller stå närmare sina förebilder inom den verkliga djurvärlden. Att vargen Zeke vill äta upp grisen Bror Duktig är till exempel ett ständigt återkommande tema, men om katten Svarte Petter skulle få för sig att äta upp musen Musse Pigg skulle detta antagligen höja ett och annat ögonbryn bland läsarna.</p>			
Nyckelord:	omsättning, kassaflöde, värdepappersmarknadslagen, yrkesutövare, intresseföretag, verifieringskedja		
Språk:	Engelska		

Acknowledgements

I wish to thank all students who use L^AT_EX for formatting their theses, because theses formatted with L^AT_EX are just so nice.

Thank you, and keep up the good work!

Espoo, June 18, 2011

Antti Ahonen

Abbreviations and Acronyms

JVM	Java Virtual Machine
JRE	Java Runtime Environment
SQA	Software Quality Assurance
AQM	Alternative Quality Model
BDD	Behavior Driven Development
TDD	Test Driven Development
ATDD	Acceptance Test Driven Development
TLD	Test Last Development
RE	Requirements Engineering
V&V	Verification & Validation
GUI	Graphical User Interface
IDE	Integrated Development Environment
DDT	Data Driven Testing
DSL	Domain Specific Language
AAA	Arrange-Act-Assert
COTM	Count of Test Methods
CC	Code Coverage
COA	Count of Assertions
COC	Count of Comments
TMNWC	Test Method Name Word Count
DDTM	Data Driven Test Methods

Contents

Abbreviations and Acronyms	vi
1 Introduction	1
1.1 General thoughts about the thesis	1
1.2 Problem statement	1
1.3 Structure of the thesis	1
2 Background	2
2.1 Java Virtual Machine	2
2.2 Software quality assurance	3
2.2.1 Definitions of quality	4
2.2.2 Motivation for SQA	4
2.2.3 Activities in SQA	5
2.3 Automated testing	6
2.3.1 Levels of automation	6
2.3.2 Low level testing explained	7
2.3.3 Benefits and drawbacks	8
2.4 Test Driven Development	9
2.4.1 Definition	9
2.4.2 Benefits and drawbacks	10
2.5 Acceptance Test Driven Development	11
2.5.1 Benefits and drawbacks	12
2.6 Behavior Driven Development	13
2.6.1 History	13
2.6.2 Definition	14
2.6.3 Process	14
2.6.4 Levels of specification	15
2.6.5 Benefits and drawbacks	18
2.7 Related research	20

3	Environment	23
3.1	xUnit testing frameworks	23
3.1.1	JUnit	23
3.1.2	Extending JUnit for Spring Framework domain	25
3.2	Implementation level BDD testing frameworks for JVM	27
3.2.1	xSpec family	27
3.2.1.1	RSpec	27
3.2.1.2	Spectrum	31
3.2.2	Gherkin family	33
3.2.2.1	Spock	33
4	Methods	37
4.1	Research questions	37
4.2	Research hypothesis	38
4.3	Empirical study	39
4.3.1	Type of study and purpose	39
4.3.2	Process for selecting teams and developers	40
4.3.3	Data collection	40
4.3.3.1	Interview for demographic purposes	41
4.3.3.2	JUnit survey	41
4.3.3.3	BDD testing frameworks survey	42
4.3.3.4	Interview about benefits and drawbacks of new BDD testing framework	43
4.3.3.5	Test code analysis	43
4.4	Validity	45
4.4.1	Threats to validity	46
4.4.1.1	Construct validity	46
4.4.1.2	Internal validity	46
4.4.1.3	External validity	46
4.4.1.4	Reliability	47
4.4.2	Improving factors of validity	47
5	BDD frameworks in selected projects	48
5.1	How project teams chose their new testing framework	48
5.1.1	Project A	49
5.1.2	Project B	53
6	Results	56
6.1	First interview: Demographics and projects	56
6.2	Surveys	56
6.3	Second interview: BDD framework feedback	59

6.4	Test code analysis	59
6.5	Comparison of BDD testing frameworks	59
7	Discussion	60
8	Conclusions	61
A	Gradle build configurations	69
B	Interview questions	73
B.1	Interview for demographic purposes	73
C	Surveys	75
C.1	Survey regarding JUnit in automated low level testing	75
C.2	Survey regarding BDD testing framework in automated low level testing	79

Chapter 1

Introduction

General thoughts about the thesis

Problem statement

Structure of the thesis

Chapter 2

Background

First in this section brief details of JVM are examined. Second, general concepts of software quality and SQA are explained. Third, SQA practice automated testing is inspected in more detail. For the rest of this chapter, agile methodologies Test Driven Development (**TDD**), Acceptance Test Driven Development (**ATDD**) and Behavior Driven Development (**BDD**) are reviewed.

Java Virtual Machine

JVM and its programming language Java was originally designed for building software on networked devices [1]. From there next major usage possibilities for Java came from Web HTML-sites. Web-sites with Java embedded programs first appeared in HotJava-browser [1]. From those days usage of Java has explored new fields and JVM itself hosts nowadays many more programming languages than just Java [2].

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 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. [1]

JVM is interesting target for a vast amount of programming languages, thanks to its maturity, ubiquity and performance [3]. These programming languages pass as a valid JVM language if their functionality can be expressed in a valid class file [2]. Programs written in ported dynamic languages such as *Ruby* (*JRuby*) or

Python (*Jython*) [3] can be run on the JVM. JVM also hosts an array of new programming languages such as *Scala*, *Clojure* and *Groovy* [2].

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.

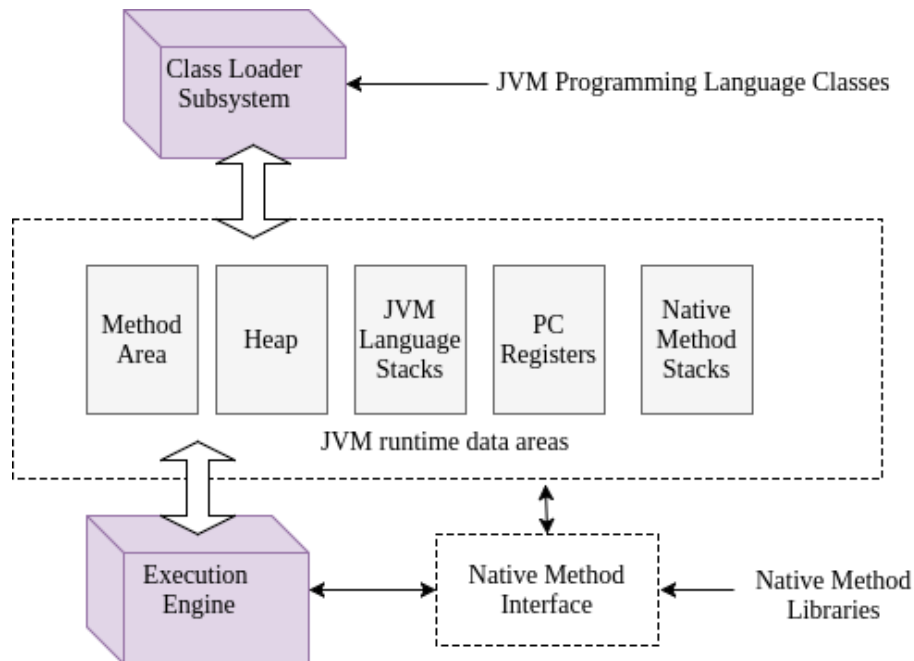


Figure 2.1: Overview of internal architecture of Java Virtual Machine

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*". [4] 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.

Definitions of quality

Quality in software development is a multifaceted entity that has had many view-points for a long time. Some of these earlier views include *product*, *transcendental*, *user*, *manufacturing* and *value based* view [5]. Because there exists so many view-points to what software quality is, it makes the measuring of quality hard [5]. Business goals and their priorities should determine the needed level of software quality [5] 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 assesment	
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

Figure 2.2: Alternative Quality Model

New alternative view on software quality by Denning [6] 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 [6]. Figure 2.2 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 [7].

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 [8]. Aggressive schedule can be usually seen as one of the primary causes for software project failure [9]. 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 [9]. Although many of the problems are related to requirements engineering (**RE**), a lot of them are fixes or rework needed after launch [10]. 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.

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 [11]. 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 [12]. 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 [12]. 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 [12]. *Continuous integration* can also be seen as a explicit defect detection as it illustrates integration defects and problems with frequent integration cycle [11].

Code enhancement activities aim to produce better design and maintainability of the codebase and these include practices like *pair programming*, *refactoring* [11] and *code reviews* [13]. 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 [12] [13].

Verification & Validation aims to guarantee the quality of product or intermediate products and it can be used for example to 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. [11]

Collaboration and communication between stakeholders are used in agile methodologies frequently being one of the cornerstones of agile practices in general [11]. Frequent customer collaboration with *On-site customer* [11] is an agile SQA-practice that establishes a foundation for many agile practices and 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**.

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.

Levels of automation

Design level		Level of automation
Requirements	→	Acceptance testing
Functional specification	→	System testing
Architecture design	→	Integration testing
Component design	→	Unit testing
↓		↑
Code		

Figure 2.3: 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 [14]. Figure 2.3 illustrates how different levels of design relate to levels of test automation in traditional **Test Last Development (TLD)**. At the bottom of figure 2.3 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.

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 [15]. 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.

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 [16]. Figure 2.3 illustrates how unit testing adheres to component design and integration testing to architecture design [14].

Unit test has many similar definitions which all agree that it tests individual unit or collection of these units working as one [17] [18]. Unit test also has the property of being isolated from the rest of the system [18]. Unit testing can in a sense also be seen as an intersection of design, coding and debugging [19]. Dedicated practitioner of unit testing Osherove [16] 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. [7]

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 [18] [16]. Osherove [16] 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 fast as unit tests [16]. Many times this results from including full context of the system through dependency injection container, such as *Spring Framework* or *Guice* [20].

Benefits and drawbacks

Benefits	Drawbacks
Rapid feedback [21]	Can't replace manual testing [22]
Improved product quality [22] [23]	Maintaining difficulty [22] [17]
Increased test coverage [22]	Lack of skilled people [22] [17]
Increased developer confidence [22]	Hard to select correct testing strategies [22] [24]
Reduced testing time [22]	Brittle tests [24]
Shorter release cycle [24]	More development time [23]
Increased testability design [24]	Cost versus value [17]
Act as documentation [19] [7] [20]	Unmaintained tests can lose all value [24]
Continuous regression [23]	

Table 2.1: 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 [21]. 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 [21]. 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. [22] 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 increase 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. [24]

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. [19] [7] [20]

At unit testing level, benefits of test automation was found to include over 20%

decrease in test defects [23]. Also the defects found by largely increased customer base in first 2 years of use was decreased by introducing unit testing practices [23]. Another strength of automated unit testing is also the continuous regression suite it provides [17].

Unfortunately test automation isn't only beneficial compared to manual testing. In the earlier mentioned review by Rafi et al. [22] 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 *inappropriate test automation strategies* [22] [24]. 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 [24]. Neglecting automated testing maintenance can lead to knowledge diminishing and tests that lose totally their capability to run [24]. 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 [24].

At unit testing level, drawbacks of test automation include approximately 30% more development time compared to manual testing [23]. Runeson [17] 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* [17]. Also the cost of unit test versus the value it provides was found problematic amongst unit testing practitioners [17].

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

Test Driven Development

Definition

Kent Beck [25] 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.

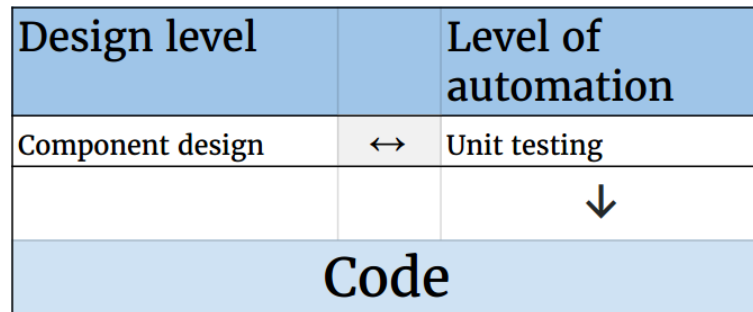


Figure 2.4: TDD related to system design

The tests written in TDD cycle are unit tests, providing incremental functionality to small pieces of software at a time [26]. Beck [27] also defines test-first principle to be much more than just testing and he states that TDD is also a software design technique. Figure 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.

Benefits and drawbacks

Benefits	Drawbacks
Increased external quality [28] [26]	Increased development time [28] [26] [23]
Increased test coverage [26]	Requires high discipline [29]
Tests executed more [23]	Can lead to brittle tests [7] [30] [31]
More precise test cases [23]	

Table 2.2: TDD Benefits & Drawbacks

Benefits of TDD are not totally clear and exact. On earlier systematic literature review by Kollanus [28] 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. [26] 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. [23] 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 [23].

Drawbacks of TDD was found unanimously to include increase in development time [28] [26] [23]. For example in the study done by Williams et al. [23] the increase was ranging from 15% to 35%. As earlier mentioned, Beck [27] defined TDD to be also a design technique. This aspect could make TDD too demanding practice for junior level developers [32].

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. [29]

BDD literature frequently states that TDD can have a tendency to shift view-point 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. [7] [30] [31]

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.

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 [33]. BDD inhods also ATDD [33], but as later is discovered, it is actually a superset of it.

ATDD is an collaboration tool for stakeholders of the project [33] [34]. It means driving the development with features specified with executable acceptance level tests [33] [34]. ATDD can have many formats, such as *Gherkin*, *Keyword-driven testing* and *Tabular formats* [33].

Instead of traditional automated testing, ATDD can be seen as a mixture of documentation centric traditional RE and communication focused agile RE [34]. The key is the communication and collaboration between stakeholders [34]. 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 [33]. Instead it is only a starting point for more quality assurance work; the system is stable enough to be manually tested further [35].

Benefits and drawbacks

Benefits	Drawbacks
Frequent collaboration [34]	Demands frequent collaboration [34]
Less ambiquity, noise and over-specification in requirements [34]	Needs active customer [34]
Faster feedback [34]	High upfront cost [34]
Increased requirement traceability [36]	Works bad with constantly changing requirements [34]
Increased trust [34]	Learning curve [34]
Increased test coverage [34]	Not for all contextes [34]
Faster start for new team members [34]	Requires high discipline [34]

Table 2.3: 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 [34] between all people involved. ATDD can potentially reduce the *ambiquity*, *noise* and *over-specification* [34] in requirements through the use of shared language. ATDD also promotes faster feedback loop [34] and traceability from requirements to code [36].

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. [34]

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. [34]

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 discipline to use properly. [34]

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

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.

History

BDD is an fairly new practice from the start of 21st century with groundwork by Dan North [37] and Dave Astels [30]. North [37] 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 [30] 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**.

Definition

BDD is seen as an evolution of TDD and ATDD [38]. It expands on the idea of driving system design with tests. North [37] brought in the concept of *ubiquitous language* to acceptance level BDD from **Domain Driven Design**. This ubiquitous language is used throughout the development lifecycle [38] and it should be executable [37]. The main reason for ubiquitous language is to build a common understanding between stakeholders [38] 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 [7].

BDD is a second generation agile methodology [7], as it incorporates into existing agile practices such as *Extreme Programming*, *Scrum* or *Kanban* [39]. It is also used iteratively & incrementally and the behavior of system should be derived from business outcomes [38]. No clear definition of BDD exists [40], but it can be seen as describing behavior of the system at all levels of granularity [7]. 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* [7]. This means doing activities just enough to incrementally provide small pieces of value, but not with the expense of quality.

Process

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

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. [7]

Value is the **driving force** to start the progress with features. To be implemented, feature must have business value related to business outcomes [37]. From there, through different levels of system, implementation is driven with tests first to specify behavior [7]. 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*

Design level		Level of automation
Business outcomes		
↓		
Requirements	↔	Stories
Functional specification	↔	Scenarios
Architecture design	↔	Integration specs
Component design	↔	Unit specs
		↓
Code		

Figure 2.5: BDD related to system development

suites [7] [39]. The outcome should be quality that delights the end user, thus performing well on the AQM [6]. Figure 2.5 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 figure 2.5, the whole automation level BDD cycle [7] needs to be explained: Figure 2.6 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 [39]. 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 progressed through all levels of behavior definition and its implementing code [7]. By repeating the steps, the end result will be a working feature providing value related to business outcomes [7].

Levels of specification

The first automation level is **acceptance level** testing [7] [38] [39] [40]. This level can be seen as a form of ATDD [33]. As earlier explained, BDD also includes the ubiquitous language [37] from Domain Driven Design and thus all ATDD does not

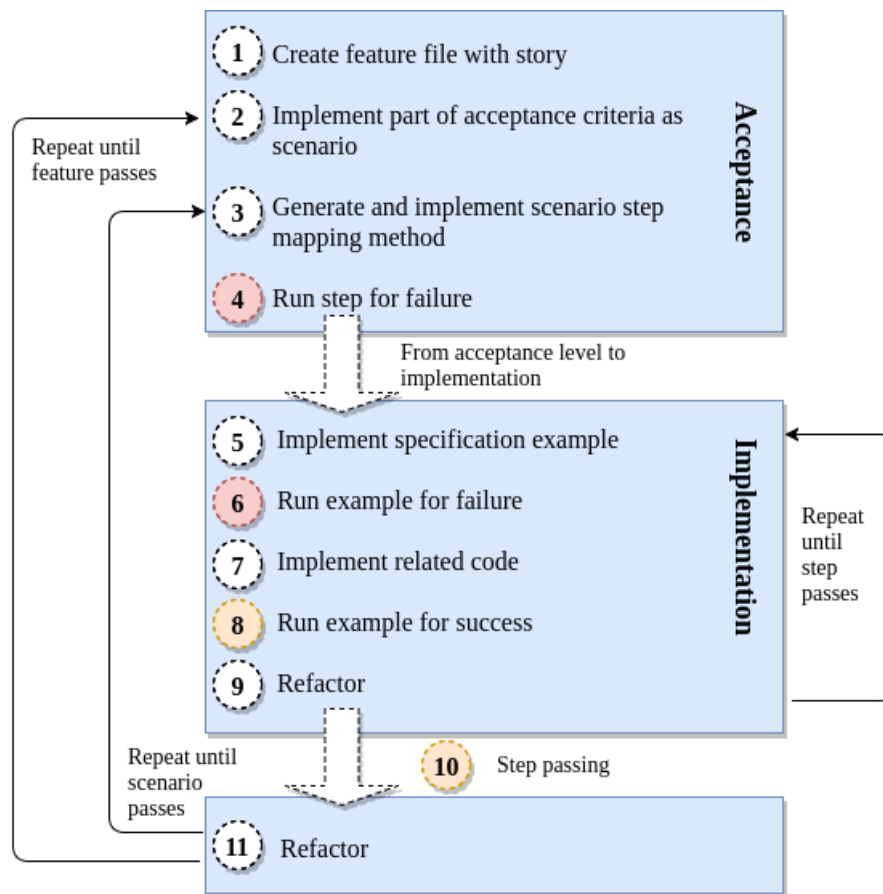


Figure 2.6: Automated BDD cycle

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 [7]. Acceptance criteria is presented as executable scenarios [37], that usually follow predetermined ubiquitous language **Gherkin** [40]. Gherkin will be studied in more detail in the next chapter when testing framework Spock is examined. Figure 2.5 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 [39].

Some of the BDD acceptance level tools for JVM include *Cucumber-JVM*, *Concordion* and *JBehave* [40]. Other environments have also BDD tools for this level, for instance *SpecFlow* for .NET and *Behave* for Python [39]. Main characteristic of acceptance level tools is business readable plain text input that is expressed

with the earlier mentioned user stories and their scenarios [40].

The second automated level in BDD is **implementation level** testing [7] [38] [39] [40]. Implementation level can be seen as testing, where behavior of objects and components are described with examples [7]. This means the earlier mentioned *unit* and *integration* testing done with a new point of view. BDD is not limited to acceptance level only, as one of the core values in it is: "*it's all behavior*" [7]. Every level can be broken down to examples describing behavior. Figure 2.5 illustrates how these implementation level specifications relate to system design. Figure 2.6 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 [39]. The tests produced by BDD differ from TDD tests, they are more granular pieces of describing examples of used code [30]. 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 [30].

The outcome from implementation level BDD specifications is readable, behavior oriented living documentation aimed for developers [7] [39]. 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 [39]. 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*.

Benefits and drawbacks

Benefits	Drawbacks
Frequent collaboration* [34]	Demands frequent collaboration* [34]
Less ambiguity, noise and over-specification in requirements* [34]	Needs active customer* [34]
Faster feedback* [34]	High upfront cost* [34]
Increased requirement traceability* [36]	Works bad with constantly changing requirements* [34]
Increased trust* [34]	Learning curve* [34]
Increased test coverage* [34]	Not for all contextes* [34]
Faster start for new team members* [34]	Requires high discipline* [34]
Ubiquitous language can help in testing right aspects [40]	
Can reduce futile feature development [39]	

Table 2.4: 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 [33]. 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 [40]. The emphasis on providing value with features can also reduce production of features that are not used [39]. **Drawbacks** of acceptance level BDD are identical to the earlier mentioned ones of ATDD.

Benefits	Drawbacks
Increased external quality* [28] [26]	Increased development time* [28] [26] [23]
Increased internal quality* [26]	Requires high discipline [39]
Helps to test right aspects of component [7] [30] [31]	
More granular test cases [7] [30]	
Can help in maintenance [39]	

Table 2.5: 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 [30]. 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 [7] [30] [31].

Implementation level BDD aims to produce more granular and descriptive test methods and test cases than TDD [7] [30]. These test cases are also describing the behavior of production code by examples [7]. This can help the system maintenance, providing up-to-date documentation for future developers or even the original developer later on in the future [39].

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 [28] [26] [23]. BDD as whole needs discipline [39], 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 [38]. 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.

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 [41], *Jasmine* in JavaScript testing [31] and *Spock* [42] 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 [43] 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 and 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. [44] 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 60.38% of practitioners found understanding of unit tests ranging from moderately difficult to very hard. Relevant findings with unit testing documentation 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 [17] 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. [23] studied effectiveness 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. [24] 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** [45] 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 [39]. 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 testing family 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 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.

xUnit 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. [46]

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

JUnit

JUnit acts as the reference implementation of the xUnit family and it is also the most popular instance of them [46]. It is used for Java-code testing and it can be extended for many domains in Java testing [46]. Current stable version of JUnit is *JUnit4* [47], but *JUnit5* is scheduled to release in Q5 of 2017 [48] providing many new features for Java testing. The examples of JUnit and empirical research in this thesis are all based on JUnit4.

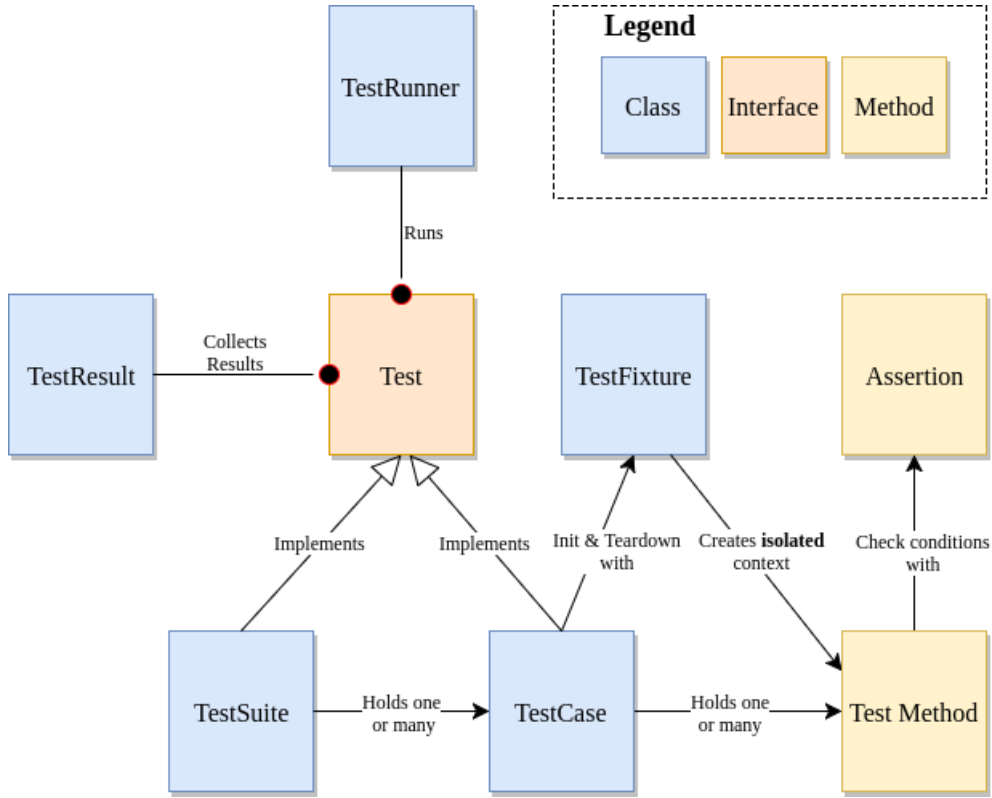


Figure 3.1: JUnit architecture

The basic architecture close to reference implementation of xUnit testing family [46] 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, but 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. [46]

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)** [20]. In the next section, example is provided of a JUnit4 test case extended for integration testing of Spring Framework domain.

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.*" [49] It is targeted for Java enterprise applications, providing teams with a framework that allows them to primarily focus on application's business logic [49]. Spring framework ships with a lot modules and features that needs to be configured for the project [50]. **Spring Boot** is *convention-over-configuration* solution composed of the Spring Framework components, that enables rapid application development with minimal effort to get started [50].

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. [51]

```
16  @RunWith(SpringJUnit4ClassRunner.class)
17  @SpringBootTest
18  public class GameServiceIntegrationTest {
19
20      @Autowired
21      private GameService gameService;
22
23      @Autowired
24      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 16 and 17. Line 16. is an example of extending JUnit with custom runner. Lines 20-21 and 23-24 show examples of **injected dependency** via Spring Framework.

Figure 3.3 displays the usage of **fixture** and two **test methods**. At lines 29-33 the fixture method inits separately before each test method run two common variables used in both tests. Lines 35-36 and 44-45 display two test method definitions with the *@Test*-annotation from JUnit4. When used, it marks the following

```

26     private GameDifficulty gameDifficulty;
27     private String playerName;
28
29     @Before
30     public void initGameValues() {
31         gameDifficulty = GameDifficulty.NORMAL;
32         playerName = "Player";
33     }
34
35     @Test
36     public void testStartGameWithNormalDifficulty() {
37         Game createdGame = gameService.startGame(playerName, gameDifficulty);
38
39         assertThat(createdGame, is(notNullValue()));
40         assertThat(createdGame.getPlayerName(), equalTo(playerName));
41         assertThat(createdGame.getDifficultyLevel(), equalTo(gameDifficulty));
42     }
43
44     @Test
45     public void testStartGamePersistsToDB() {
46         Integer gameCountBeforeStartGame = gameRepository.findAll().size();
47
48         gameService.startGame(playerName, gameDifficulty);
49
50         Integer gameCountAfterStartGame = gameRepository.findAll().size();
51         assertThat(gameCountAfterStartGame, is(greaterThan(gameCountBeforeStartGame)));
52     }

```

Figure 3.3: JUnit Spring integration tests

method as a runnable test method. Both test methods contain **assertions**. Assertions in use at lines 39-41 and 51 are *Hamcrest matchers* [52], that allows more readable descriptions for used assertions than traditional JUnit assertions.

GameServiceIntegrationTest (fi.aalto.ekanban.services)	268ms
testStartGameWithNormalDifficulty	166ms
testStartGamePersistsToDB	102ms

Figure 3.4: JUnit test run results

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 [39].

Build configuration of JUnit4 used in examples can be found in Appendix A figure A.1. Used build tool is Groovy-based *Gradle* [53].

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.

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 to practicing BDD: **xSpec family** [38] and **Gherkin family** testing frameworks.

xSpec family

xSpec family testing frameworks are restricted to implementation level [38]. 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 environment, one of them being for example Scala-based **FunSpec** [54] for ScalaTest, but it is not examined in detail.

RSpec

Ruby community has been for a long time a steady advocate of TDD [41]. It is also the birth place of first implementation level BDD testing framework: RSpec [30]. Whereas first BDD framework JBehave [37] was aimed for acceptance level and all stakeholders, RSpec brought the behavior driven thinking for developer audience [30].

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* [55]. Later on in this section, RSpec usage for testing Java-code both at unit and integration level for Spring framework is reviewed. Before this, the core concepts of RSpec are illustrated.

The main functionality of RSpec comes from the *rspec-core* library [55]. It includes the code **example groups** that can hold runnable specification **code examples** [7]. These examples and groups are created into a **spec** file [7]. Example groups can be initialized with keywords *describe* and *context*, which both are aliases for each other [56]. Example groups can inherit each other nested in a single file, creating nested context groups [56]. This can help immensely on removing

repetition from the test code and achieving easily readable test outputs [7]. This is enabled by easily usable **lifecycle hooks**, such as *before*, *after* and *around* [7]. 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* [56]. These examples create executable pieces of behavior of the code under specification [7]. Code examples contain **expectations**, which specify the expected behavior of the given example [7]. Expectations can be seen as assertions from xUnit testing family, but the reason for changing the language is to support better communication between stakeholders, in this case developers [7]. 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 [30]. The guideline for the code examples is to hold only one expectation per example [7]. This allows separate info about failing situations [7] and provides a rule **one assert per test method** for living specification [30]. This rule was originally created by Astels [30] 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 [7]:

Expectations	⇒ Assertions
Code Example	⇒ Test Method
Example Group	⇒ Test Case
Spec File	⇒ Test Suite
Lifecycle Hook	⇒ Test Fixture

```

16 describe GameService do
17
18   before(:all) do
19     @gameInitService = Mockito.mock(GameInitService.java_class)
20     @playerService = Mockito.mock(PlayerService.java_class)
21     @gameOptionService = Mockito.mock(GameOptionService.java_class)
22     @gameRepository = Mockito.mock(GameRepository.java_class)
23     @game_service = GameService.new(@gameInitService, @playerService,
24                                     @gameOptionService, @gameRepository)
25   end

```

Figure 3.5: RSpec 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 16. shows the creation of example group with the keyword *describe*. Lines from 18 to 25 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** [57] is used for creating test double objects that replace the dependencies of `GameService`.

Figure 3.6 illustrates use of nested example groups that contains two examples in the second level nested example group. Line 27. is the first example group, that is created with keyword *describe*. The second level nested example group starts at line 36. 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. RSpec lets to define the **action** of the

```

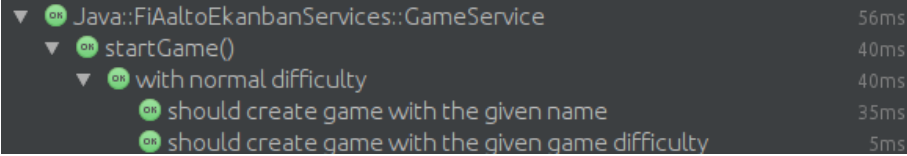
27 describe 'startGame()' do
28
29   let(:player_name) {"player"}
30   let(:game_difficulty) {GameDifficulty::NORMAL}
31   let(:new_game) {GameBuilder.aGame().
32                     with_player_name(player_name).
33                     with_difficulty_level(game_difficulty).
34                     build()}
35
36   context 'with normal difficulty' do
37
38     before(:each) do
39       Mockito.when(@gameInitService.getInitializedGame(
40         Mockito.any(GameDifficulty.java_class), Mockito.anyString)).thenReturn(new_game)
41       Mockito.when(@gameRepository.save(new_game)).thenReturn(new_game)
42     end
43
44     subject(:created_game) { @game_service.start_game(player_name, game_difficulty) }
45
46     it 'should create game with the given name' do
47       expect(created_game.player_name).to eq player_name
48     end
49
50     it 'should create game with the given game difficulty' do
51       expect(created_game.difficulty_level).to eq game_difficulty
52     end
53
54   end
55
56 end

```

Figure 3.6: RSpec nested example groups with code examples

example group with the keyword *subject* [58]. Example of used subject is at line 44. The two code examples are at lines 46-48 and 50-52. Both examples show **expectations** given with keyword *expect* at lines 47 and 51. 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.

Figure 3.7 shows the output of code examples run in Integrated Development Environment (IDE) *Intellij Idea* [59]. 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 first code example is formatted as



▼	OK	Java::FiAaltoEkanbanServices::GameService	56ms
▼	OK	startGame()	40ms
▼	OK	with normal difficulty	40ms
	OK	should create game with the given name	35ms
	OK	should create game with the given game difficulty	5ms

Figure 3.7: RSpec code example run output

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

```

9  class SpringContext
10   include Singleton
11
12   def initialize
13     @ctx = AnnotationConfigEmbeddedWebApplicationContext.new
14     @ctx.scan("fi.aalto.ekanban")
15     @ctx.getEnvironment.setActiveProfiles("test")
16     @ctx.register(MongoConfiguration.java_class)
17     @ctx.register(PortConfiguration.java_class)
18     @ctx.refresh
19   end
20
21   def spring_ctx
22     @ctx
23   end
24
25   def spring_port
26     @ctx.getEmbeddedServletContainer.getPort
27   end
28
29 end

```

Figure 3.8: RSpec extension for Spring Framework integration testing

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 14, it scans the given package for dependencies. At lines 15-17 test specific configuration is registered for the test context. Lines 25-27 display how to retrieve web server port for 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.

```

10 describe GameService do
11
12   before(:all) do
13     @ctx = SpringContext.instance.spring_ctx
14     @gameRepository = @ctx.getBean "gameRepository"
15   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 A figure A.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* [60] (dependencies) installed is required.

Spectrum

Spectrum is a Java 8 -based BDD-style **test runner for JUnit 4**. 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. [61]

Spectrum used in examples and later on in projects 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*. [62]

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.


```

44 describe("startGame", () -> {
45
46     final Supplier<GameDifficulty> gameDifficulty = let(() -> GameDifficulty.NORMAL);
47     final Supplier<String> playerName = let(() -> "Player");
48     final Supplier<Game> newGame = let(() -> GameBuilder.aGame()
49         .withPlayerName(playerName.get())
50         .withDifficultyLevel(gameDifficulty.get())
51         .build());
52
53     describe("with normal difficulty", () -> {
54
55         beforeEach(() -> {
56             Mockito.when(gameInitService.getInitializedGame(
57                 Mockito.any(GameDifficulty.class),
58                 Mockito.any(String.class))).thenReturn(newGame.get());
59             Mockito.when(gameRepository.save(Mockito.any(Game.class))).thenReturn(newGame.get());
60         });
61
62         final Supplier<Game> createdGame = let(() ->
63             gameService.startGame(playerName.get(), gameDifficulty.get()));
64
65         it("should create game with the given name", () -> {
66             assertThat(createdGame.get().getPlayerName(), equalTo(playerName.get()));
67         });
68
69         it("should create game with the given game difficulty", () -> {
70             assertThat(createdGame.get().getDifficultyLevel(), equalTo(gameDifficulty.get()));
71         });
72     });
73 });
74

```

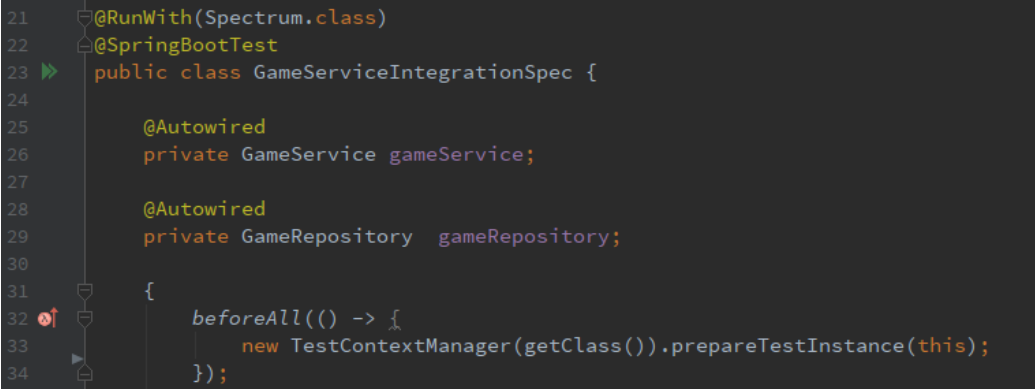
Figure 3.10: Spectrum nested example groups with code examples

GameServiceSpec (fi.aalto.ekanban.services)	373ms
GameService	373ms
startGame	373ms
with normal difficulty	373ms
should create game with the given name	372ms
should create game with the given game difficulty	1ms

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 22 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 33 displays alternative imperative way of loading test context. Lines 25-26 and 28-29 display the dependency injection feature of Spring Framework.

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



```

21  @RunWith(Spectrum.class)
22  @SpringBootTest
23  public class GameServiceIntegrationSpec {
24
25      @Autowired
26      private GameService gameService;
27
28      @Autowired
29      private GameRepository gameRepository;
30
31      {
32          beforeAll(() -> {
33              new TestContextManager(getClass()).prepareTestInstance(this);
34          });
35      }
36  }

```

Figure 3.12: Spectrum extension for Spring Framework integration testing

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.

Gherkin family

Gherkin family is a term defined in this thesis, containing BDD testing frameworks that use predetermined ubiquitous language **Gherkin** [63]. There exists research related to usage of Gherkin in BDD testing frameworks [40]. Gherkin family frameworks exists for both acceptance and implementation testing level [40]. For Java-code testing at implementation level with Gherkin, JVM offers for example **Spock** [42] via Groovy, Java 8 -based **Spectrum** [61], JRuby-based **RSpec-given** [64] for Rspec and Scala-based **FeatureSpec** [65] for ScalaTest. Spock will be used as an example for inspecting implementation level testing with a Gherkin family framework.

Spock

Spock is a BDD testing framework for JVM programming language Groovy [20]. It supports testing both Java and Groovy code and it can be considered a superset of JUnit, as it extends the JUnit runner [42]. 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* [20].

Spock is capable of testing the whole automated BDD cycle from acceptance level to unit level [20]. It holds unit test support with integrated mocking and stub-

bing capabilities [42]. Spock can also be extended for integration testing of different domains [20]. Spring Framework domain extension is done with *spock-spring* dependency, that can be configured with the *@ContextConfiguration*-annotation of Spring [51].

Spock also supports **DDT** with tabular format readable domain specific language (DSL) [42]. DDT can drastically remove repetition from test code and makes it easier to test different parameter variations [20]. 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 [20].

```
Condition not satisfied:

createdGame.difficultyLevel != gameDifficulty
|                               | |
|                               | |
|           NORMAL            | NORMAL
|                               |
|                               false
fi.aalto.ekanban.models.db.games.Game@3c5ec58b
```

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 for plain text test input [63]. 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 [66]. 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 [40].

Spock terminology consists of **specification** files containing **feature methods** and Given-When-Then **blocks** inside them [42]. Figure 3.14 displays initialization of a Spock specification file. At line 12, the specification class is created by extending the class *Spock.lang.Specification*. Lines 14-16 show use of *@Shared*-variables, which are used for long lived objects shared between feature methods [42]. Lines 18 to 24 show the use of *setup*-fixture for creating a shared, isolated context for each feature method [42]. The fixture setup is used for initializing unit testing context, where the use of *Mock*-objects [42] 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 [20]:

- **Given** is a code block used to initialize the **context** of test

```

12 class GameServiceSpockSpec extends Specification {
13
14     @Shared GameService gameService
15     @Shared GameInitService gameInitService
16     @Shared GameRepository gameRepository
17
18     def setup() {
19         gameInitService = Mock(GameInitService)
20         gameRepository = Mock(GameRepository)
21         def playerService = Mock(PlayerService)
22         def gameOptionService = Mock(GameOptionService)
23         gameService = new GameService(gameInitService, playerService,
24                                     gameOptionService, gameRepository)
25     }

```

Figure 3.14: Spock specification file initialization

- **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 [19]. As this structure is not enforced in JUnit, there is the considerable possibility that it will not be used [20].

Figure 3.15 illustrates the contents of a feature method and the use of Given-When-Then blocks. Block definitions are at lines 29, 35, 38, 42, 45 and 48 together with optional [42] textual descriptions. Lines 29-33 form a *Setup*-block, which is an alias for Given-block [42]. Lines 42 and 45 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 [20].

Lines 39-40 display the verifying of mock object actions and stubbing capabilities. For example, at line 39

```
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 26 and 48-51. Line 26 *@Unroll* creates a separate test run for each parameter variation defined in the *Where*-block (lines 48-51). Line 49. defines the parameter names and lines 50-51 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 [20]. In the example,

```

26 @Unroll
27 def "GameService startGame() with playerName #playerName and difficulty #gameDifficulty"() {
28
29     setup:
30         def newGame = GameBuilder.aGame()
31             .withPlayerName(playerName)
32             .withDifficultyLevel(gameDifficulty)
33             .build()
34
35     when: "startGame() is called with playerName and gameDifficulty"
36         def createdGame = gameService.startGame(playerName, gameDifficulty)
37
38     then: "game should be initialized and persisted"
39         1 * gameInitService.getInitializedGame(gameDifficulty, playerName) >> newGame
40         1 * gameRepository.save(newGame) >> newGame
41
42     and: "game has been created with given name"
43         createdGame.playerName == playerName
44
45     and: "game has been created with given gameDifficulty"
46         createdGame.difficultyLevel != gameDifficulty
47
48     where:
49         playerName | gameDifficulty
50         "Player"   | GameDifficulty.NORMAL
51         "Pelaaja"  | GameDifficulty.NORMAL
52
53 }

```

Figure 3.15: Spock feature method

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 3.15 as two separate test runs through the DDT.

GameServiceSpockSpec (fi.aalto.ekanban.services)	533ms
GameService startGame() with playerName Player and difficulty NORMAL	521ms
GameService startGame() with playerName Pelaaja and difficulty NORMAL	12ms

Figure 3.16: Spock feature method output

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 A figure A.3.

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 [20]. 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 empirical 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 and features of studied testing frameworks explained in chapter 3. Third, the empirical study and its methodology are explained in detail. Finally, limitations to the study methods are discussed.

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:

1. **RQ1:** How does behavior driven testing frameworks change developer practices working with automated low level testing compared to JUnit framework?
2. **RQ2:** How does behavior driven testing frameworks change developer perception of working with automated low level testing compared to JUnit framework?
3. **RQ3:** How does behavior driven testing frameworks change written low level test cases and test code coverage compared to JUnit framework?

Research hypothesis

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

Statement 1: 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 [7] [30]. 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 [30]. Compared to earlier average in the project, statement 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.

Statement 2: 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 [7] [30] [31]. 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 [39]. 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 [45]. Statement is measured with a survey and interview.

Statement 3: 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 [39]. Especially the maintaining of test cases should be easier with repetition reducing techniques provided by these BDD testing frameworks [7] [20]. 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. Statement is measured with a survey and interview.

Statement 4: Developers will perceive working with low level automated testing more enjoyable.

This statement is a combination of the three earlier statements. 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 statements 2 and 3, easier understanding and maintaining of test cases should affect how automated low level testing is perceived. Statement is measured with a survey and interview.

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, threats to validity and limitations in this study are reviewed.

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 [67] and case study research [68].

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 testing frameworks and implementation level BDD testing frameworks for larger scale studies.

Objective of the study was to compare traditional xUnit testing frameworks and implementation level BDD testing frameworks and how they change the de-

veloper 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.

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 ???. 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, and how developers chose which BDD testing framework to take into use, are explained in more detail in the next chapter.

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 quantitative 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.

Interview for demographic purposes

Participant demographics were studied with recorded semi-structured interviews [69]. The interview was constructed following the guidelines of suggested best practices [67]. The results of interviews and the analyzed demographics are shown in the chapter 6. The questions of the interview can be found from Appendix B section B.1.

JUnit survey

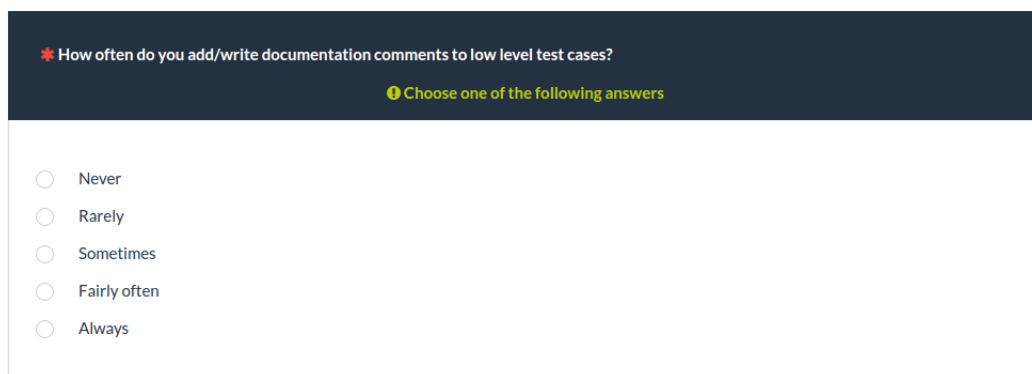
An online survey about JUnit was built to gather base information for RQ1 and RQ2, that was later on used as a reference to answer about changes in automated low level testing practices and perception. The survey was designed and built using *LimeSurvey* [70]. The reason to use survey as the main tool for data collection was the related unit testing research surveys [23] [43] [44]. Some of the survey questions were interesting starting points for the built survey in this study. As there were many survey questions, 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 [71]. At the end of the survey **Network Promoter Score (NPS)** [72] was used to see how enthusiastic 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 implementation of survey, its questions and how it relates to related research survey questions can be found from Appendix...

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.

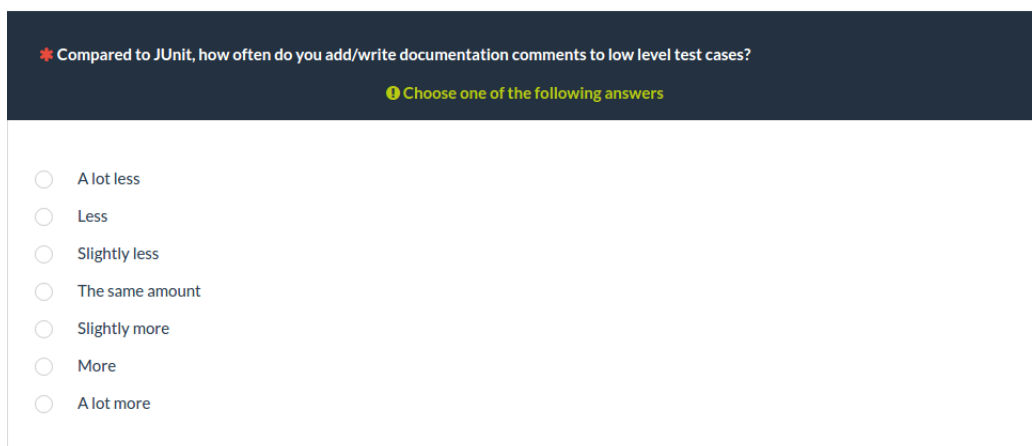


★ How often do you add/write documentation comments to low level test cases?

📌 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 implementation of survey, its questions and how it relates to related research survey questions can be found from Appendix...

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 two 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?

The idea to practice these two 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. Framework X part of the question relates to chosen BDD testing framework in the project, as it was not the same for both projects.

Test code analysis

Research question 3 is answered with test code analysis done with observations 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 values are then compared against each other, to see how the following aspects of test code have changed:

Automated unit testing level

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).

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

2. *Code coverage (CC)*: Measured code coverage for unit tests is done with *JaCoCo* [73]. The code coverage is measured with **instruction coverage** and **branch coverage** percentages [74]. 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 levels

3. *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).

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

4. *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.

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

5. *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* [75] 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 figure 3.6 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).

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

6. *Data driven test methods (DDTM)*: DDTM, Ratio of data driven low level test methods (DDLLTM) to standard low level test methods (SLLTM) is calculated by dividing the sum of DDLLTM with the sum of SLLTM.

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

Validity

In this section, first the threats to validity of the study are analyzed. After that, the improving factors of validity are discussed.

Threats to validity

Threats to validity are categorized by aspects recommended by Runeson et al. [68]. Relevant threat categories in this research include *construct*, *internal* & *external validity* and *reliability* of the study.

Construct validity

The construct validity threat includes the threats to validity of the surveys and interviews. Although there was support available for the participants during the answering of the survey to avoid misunderstood questions, there is 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.

Internal validity

Internal validity threat includes the unexpected sources of bias [67]. As I as the researcher in this study have a reasonable long history with BDD testing frameworks, it can result in too much enthusiasm in the introduction of new testing framework for the developer participants. As one of the key characteristics of company's developers is the high autonomy in projects, 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 section OOOO, 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**".

External validity

External validity threats are related to threats that hinder the possibility to generalize the findings from this study [68]. The demographics which the study was applied can be found in section XXXXX. 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 YYYY, 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 developer working, but can't be used to state a generalizations of unit and integration testing practices. 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** and generalizations to other environments can't be made properly. 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.

Reliability

Reliability of this study is the aspect of the study being dependent on the researcher [68]. The study data collection methods are explained in detail and thus they should be able to be replicated by other researchers. One aspect that hinders the reliability is the provided refactoring of JUnit tests to BDD testing framework examples explained in section 5.1. 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.

Improving factors of validity

Improving factors for the validity of this case study can be categorized under *prolonged involvement* and *triangulation* [68]. Prolonged involvement was achieved by the employment of the thesis worker, researcher, 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. The risks of this involvement were analyzed as the possible bias by presence of a "champion".

Triangulation was used to increase the validity of the study. It was visible through **data triangulation** by using multiple projects and their developers as the data source. **Methodological triangulation** was used in combining interviews, surveys and code analysis in data collection of the study. Large part of survey regarding the developer practices and perception towards JUnit were well-founded by using previous research survey questions as the starting base directly.

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 different BDD testing frameworks.

How project teams chose their new testing framework

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 project 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.

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.

```

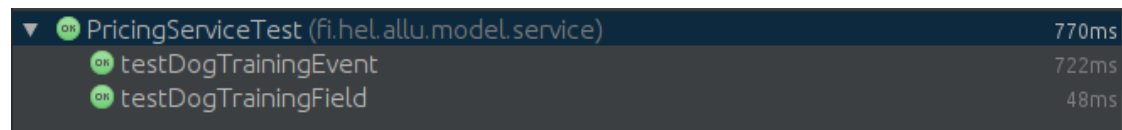
128  @Test
129  public void testDogTrainingEvent() {
130      Application application = new Application();
131      application.setType(ApplicationType.SHORT_TERM_RENTAL);
132      application.setKind(ApplicationKind.DOG_TRAINING_EVENT);
133      application.setStartTime(ZonedDateTime.parse("2016-11-07T06:00:00+02:00"));
134      application.setEndTime(ZonedDateTime.parse("2016-12-10T05:59:59+02:00"));
135      Applicant applicant = new Applicant();
136      applicant.setName("Hakija");
137      applicant.setType(ApplicantType.ASSOCIATION);
138      application.setApplicantId(applicantDao.insert(applicant).getId());
139      // association -> 50 EUR /applicationExtension
140      List<InvoiceRow> invoiceRows = new ArrayList<>();
141      pricingService.updatePrice(application, invoiceRows);
142      assertEquals(5000, application.getCalculatedPrice().intValue());
143
144      applicant.setType(ApplicantType.COMPANY);
145      application.setApplicantId(applicantDao.insert(applicant).getId());
146      // Company -> 100 EUR /applicationExtension
147      invoiceRows.clear();
148      pricingService.updatePrice(application, invoiceRows);
149      assertEquals(10000, application.getCalculatedPrice().intValue());
150  }
151
152  @Test
153  public void testDogTrainingField() {
154      Application application = new Application();
155      application.setType(ApplicationType.SHORT_TERM_RENTAL);
156      application.setKind(ApplicationKind.DOG_TRAINING_FIELD);
157      application.setStartTime(ZonedDateTime.parse("2016-11-07T06:00:00+02:00"));
158      application.setEndTime(ZonedDateTime.parse("2016-12-10T05:59:59+02:00"));
159      Applicant applicant = new Applicant();
160      applicant.setName("Hakija");
161      applicant.setType(ApplicantType.ASSOCIATION);
162      application.setApplicantId(applicantDao.insert(applicant).getId());
163      // association -> 100 EUR /year
164      List<InvoiceRow> invoiceRows = new ArrayList<>();
165      pricingService.updatePrice(application, invoiceRows);
166      assertEquals(30000, application.getCalculatedPrice().intValue());
167
168      applicant.setType(ApplicantType.COMPANY);
169      application.setApplicantId(applicantDao.insert(applicant).getId());
170      // Company -> 200 EUR /year
171      invoiceRows.clear();
172      pricingService.updatePrice(application, invoiceRows);
173      assertEquals(60000, application.getCalculatedPrice().intValue());
174  }

```

Figure 5.1: JUnit test methods to be refactored

Figure 5.1 displays the example JUnit test methods. Their test run output is display 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 130-142 and lines 144-147 are using the same *context* of the test with modifications for a new test. At line 148 is the *stimulus* of the new test and 149 holds the new *assertion* made. Both of these test methods share mostly the same test method code and thus figures 5.3 and 5.5 display them refactored into tests with Spock and Spectrum that produce separate test runs with minimized repeated code.



▼ OR PricingServiceTest (fi.hel.allu.model.service)	770ms
OR testDogTrainingEvent	722ms
OR testDogTrainingField	48ms

Figure 5.2: JUnit test method run outputs

```

93  @Unroll("getCalculatedPrice() with kind of #kind, rental time of #rentTime and applicant is #applicantType" +
94      " should amount to #euroAmount euros")
95  def "Application getCalculatedPrice() with applicant"() {
96
97      given: "application with kind #kind and rent time of #rentTime"
98          def startTime = "2016-11-07T06:00:00+02:00"
99          initApplicationWithGivenProperties(kind, startTime, endTime)
100      and: "application has an applicant of type #applicantType"
101          def applicant = new Applicant()
102          applicant.setName("Hakija")
103          applicant.setType(applicantType)
104          application.setApplicantId(applicantDao.insert(applicant).getId())
105
106      when: "price is updated for the given application"
107          pricingService.updatePrice(application, invoiceRows)
108
109      then: "calculated price should be #euroAmount euros"
110          application.getCalculatedPrice().intValue() == intAmount
111
112      where:
113          kind | rentTime | euroAmount | intAmount | applicantType | _
114          ApplicationKind.DOG_TRAINING_EVENT | "33 days" | "50" | 5000 | ApplicantType.ASSOCIATION | _
115          ApplicationKind.DOG_TRAINING_EVENT | "33 days" | "100" | 10000 | ApplicantType.COMPANY | _
116          ApplicationKind.DOG_TRAINING_FIELD | "3 years" | "300" | 30000 | ApplicantType.ASSOCIATION | _
117          ApplicationKind.DOG_TRAINING_FIELD | "3 years" | "600" | 60000 | ApplicantType.COMPANY | _
118
119      endTime << [
120          "2016-12-10T05:59:59+02:00",
121          "2016-12-10T05:59:59+02:00",
122          "2018-12-10T05:59:59+02:00",
123          "2018-12-10T05:59:59+02:00"
124      ]
125  }
126

```

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

In figure 5.3, lines 112-124 together with lines 93-94 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.

Test Case	Duration
getCalculatedPrice() with kind of DOG_TRAINING_EVENT, rental time of 33 days and applicant is ASSOCIATION should amount to 50 euros	65 933ms
getCalculatedPrice() with kind of DOG_TRAINING_EVENT, rental time of 33 days and applicant is COMPANY should amount to 100 euros	23ms
getCalculatedPrice() with kind of DOG_TRAINING_FIELD, rental time of 3 years and applicant is ASSOCIATION should amount to 300 euros	20ms
getCalculatedPrice() with kind of DOG_TRAINING_FIELD, rental time of 3 years and applicant is COMPANY should amount to 600 euros	21ms

Figure 5.4: Spock refactored example test run output

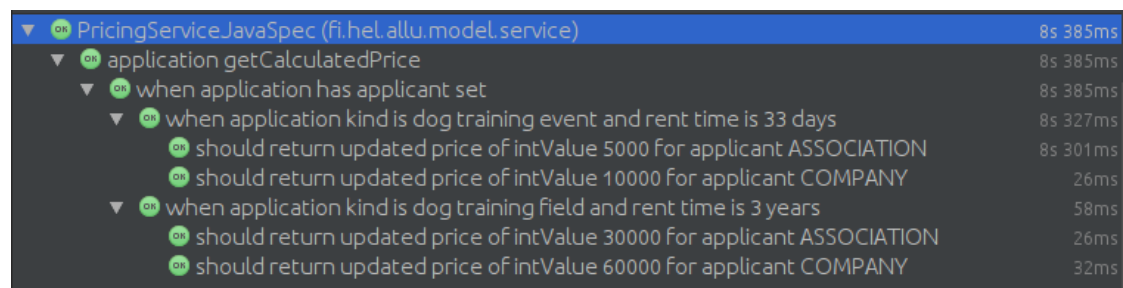
```

59 describe("application getCalculatedPrice", () -> {
60
61     beforeAll(() -> {
62         startTime = "2017-06-15T08:30:00+02:00";
63     });
64
65     describe("when application has applicant set", () -> {
66         describe("when application kind is dog training event and rent time is 33 days", () -> {
67             beforeEach(() -> {
68                 endTime = "2017-07-17T08:30:00+02:00";
69                 initApplicationWithGivenProperties(ApplicationKind.DOG_TRAINING_EVENT, startTime, endTime);
70             });
71             Arrays.asList(makePair(ApplicantType.ASSOCIATION, 5000),
72                 makePair(ApplicantType.COMPANY, 10000)).forEach(applicant -> {
73                 it("should return updated price of intValue " + applicant.getValue()
74                     + " for applicant " + applicant.getKey(), () -> {
75                     updateApplicationPriceForGivenApplicantType(applicant.getKey());
76                     updatedPrice = application.getCalculatedPrice();
77                     assertEquals(updatedPrice, applicant.getValue());
78                 });
79             });
80         });
81         describe("when application kind is dog training field and rent time is 3 years", () -> {
82             beforeEach(() -> {
83                 endTime = "2019-10-15T08:30:00+02:00";
84                 initApplicationWithGivenProperties(ApplicationKind.DOG_TRAINING_FIELD, startTime, endTime);
85             });
86             Arrays.asList(makePair(ApplicantType.ASSOCIATION, 30000),
87                 makePair(ApplicantType.COMPANY, 60000)).forEach(applicant -> {
88                 it("should return updated price of intValue " + applicant.getValue()
89                     + " for applicant " + applicant.getKey(), () -> {
90                     updateApplicationPriceForGivenApplicantType(applicant.getKey());
91                     updatedPrice = application.getCalculatedPrice();
92                     assertEquals(updatedPrice, applicant.getValue());
93                 });
94             });
95         });
96     });
97 });

```

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 71-79 and 86-94 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 concatenated into test output that can be seen in figure 5.6.

A screenshot of a test runner's output window showing a hierarchical tree of test results. The root node is 'PricingServiceJavaSpec (fi.hel.allu.model.service)' with a green 'OK' icon and a duration of 8s 385ms. It has a child 'application getCalculatedPrice' (8s 385ms), which in turn has a child 'when application has applicant set' (8s 385ms). This 'when' block contains two 'should' blocks: 'should return updated price of intValue 5000 for applicant ASSOCIATION' (8s 301ms) and 'should return updated price of intValue 10000 for applicant COMPANY' (26ms). The 'should' blocks have green 'OK' icons. The 'when' block also has a child 'when application kind is dog training field and rent time is 3 years' (58ms), which contains two 'should' blocks: 'should return updated price of intValue 30000 for applicant ASSOCIATION' (26ms) and 'should return updated price of intValue 60000 for applicant COMPANY' (32ms). All 'should' blocks also have green 'OK' icons.

▼ OK PricingServiceJavaSpec (fi.hel.allu.model.service)	8s 385ms
▼ OK application getCalculatedPrice	8s 385ms
▼ OK when application has applicant set	8s 385ms
▼ OK when application kind is dog training event and rent time is 33 days	8s 327ms
OK should return updated price of intValue 5000 for applicant ASSOCIATION	8s 301ms
OK should return updated price of intValue 10000 for applicant COMPANY	26ms
▼ OK when application kind is dog training field and rent time is 3 years	58ms
OK should return updated price of intValue 30000 for applicant ASSOCIATION	26ms
OK should return updated price of intValue 60000 for applicant COMPANY	32ms

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 where 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 of the main reasons why developers chose it over Spock.

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.

```

13  @RunWith(Parameterized.class)
14  public class NameValidatorTest {
15      static final int MAX_CHARACTERS = 300;
16
17      static final String MSG_VALID = "";
18      static final String MSG_INVALID_LENGTH = "fi.yincit.validation.Name.invalidLength";
19      static final String MSG_INVALID_FIRST_CHAR = "fi.yincit.validation.Name.invalidFirstChar";
20      static final String MSG_INVALID_CHAR = "fi.yincit.validation.Name.invalidChar";
21
22      private String name;
23      private boolean isValid;
24      private String message;
25
26      public NameValidatorTest(String name, boolean isValid, String message) {
27          this.name = name;
28          this.isValid = isValid;
29          this.message = message;
30      }
31
32      @Parameters(name = "{index}: name: \"{0}\"")
33      public static Collection<Object[]> generateData() {
34          char[] maxLength = new char[MAX_CHARACTERS];
35          char[] tooLong = new char[MAX_CHARACTERS + 1];
36          Arrays.fill(maxLength, 'a');
37          Arrays.fill(tooLong, 'a');
38          Object[][] values = new Object[][]{
39              {null, false, MSG_INVALID_LENGTH},
40              {"", false, MSG_INVALID_LENGTH},
41              {"Valid name", true, MSG_VALID},
42              {"Invalid character!", false, MSG_INVALID_CHAR},
43              {"- is invalid start character", false, MSG_INVALID_FIRST_CHAR},
44              {"_ is invalid start character", false, MSG_INVALID_FIRST_CHAR},
45              {"\\ is invalid start character", false, MSG_INVALID_FIRST_CHAR},
46              {"/ is invalid start character", false, MSG_INVALID_FIRST_CHAR},
47              {"( is invalid start character", false, MSG_INVALID_FIRST_CHAR},
48              {") is invalid start character", false, MSG_INVALID_FIRST_CHAR},
49              {"& is invalid start character", false, MSG_INVALID_FIRST_CHAR},
50              {"' is invalid start character", false, MSG_INVALID_FIRST_CHAR},
51              {"+ is invalid start character", false, MSG_INVALID_FIRST_CHAR},
52              {"a ok but ^ is invalid character", false, MSG_INVALID_CHAR},
53              {"@ ok but ? is invalid character", false, MSG_INVALID_CHAR},
54              {"@ Should work", true, MSG_VALID},
55              {"! Should work", true, MSG_VALID},
56              {"Valid special chars ÆÄÅö_@./\\()&'!+", true, MSG_VALID},
57              {"aa", true, MSG_VALID},
58              {"a", false, MSG_INVALID_LENGTH},
59              {new String(maxLength), true, MSG_VALID},
60              {new String(tooLong), false, MSG_INVALID_LENGTH},
61          };
62          return Arrays.asList(values);
63      }
64
65      @Test
66      public void testNames() throws Exception {
67          NameValidator validator = new NameValidator();
68
69          assertEquals(validator.isValid(name, null), isValid);
70          assertEquals(validator.getErrorMessage(), message);
71      }
72  }
73  }

```

Figure 5.7: JUnit DDT example

Figure 5.7 displays the chosen JUnit test case for refactoring. At line 13, JUnit is extended with *Parameterized* custom runner [76], which adds the DDT support for JUnit. Lines 26-30 and 32-63 display the creation of DDT test case setup in this JUnit example. At lines 65-71 is test method which uses this DDT setup. The result of running the test case can be seen in figure 5.10.

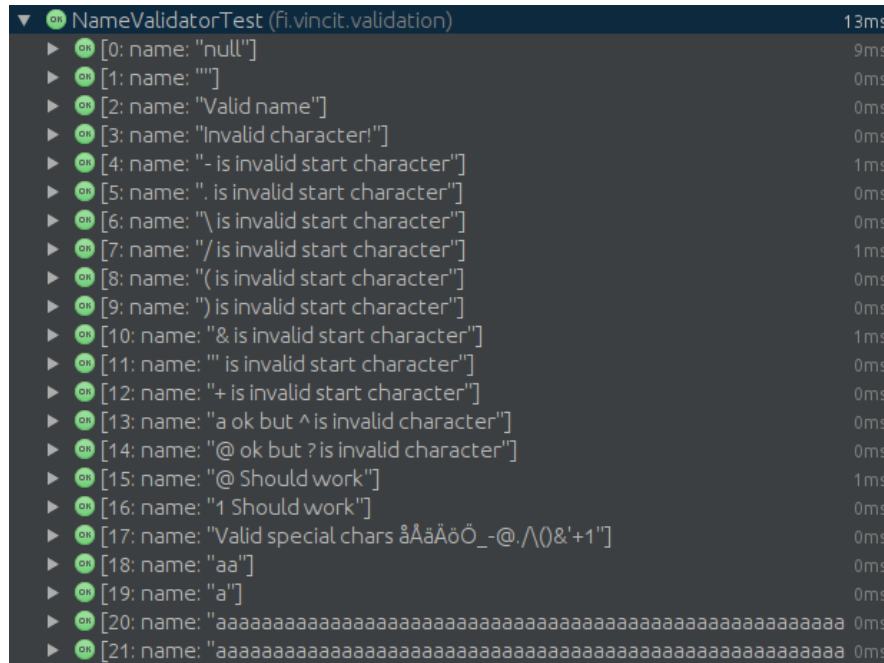


Figure 5.8: JUnit DDT example test run output

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 37-60 the DDT table format is defined with *where*-block and at line 24 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.

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.

```

7 class NameValidatorSpec extends Specification {
8
9     static final int MAX_CHARACTERS = 300
10
11     static final String MSG_VALID = ""
12     static final String MSG_INVALID_LENGTH = "fi.vincit.validation.Name.invalidLength"
13     static final String MSG_INVALID_FIRST_CHAR = "fi.vincit.validation.Name.invalidFirstChar"
14     static final String MSG_INVALID_CHAR = "fi.vincit.validation.Name.invalidChar"
15
16     @Shared char[] maxLength
17     @Shared char[] tooLong
18
19     def setupSpec() {
20         maxLength = ["a"] * MAX_CHARACTERS
21         tooLong = ["a"] * (MAX_CHARACTERS + 1)
22     }
23
24     @Unroll("validating name 'name' should be #result as valid")
25     def "name validator" () {
26         given: "a new name validator"
27             def nameValidator = new NameValidator()
28
29         when: "given #name name is validated"
30             def nameIsValid = nameValidator.isValid(name, null)
31
32         then: "name should be #result as valid"
33             nameIsValid == resultBoolean
34         and: "validator should have produced #message error message #actualMessage"
35             nameValidator.errorMessage == actualMessage
36
37         where:
38             name | result | resultBoolean | message | actualMessage
39             "Valid name" | "accepted" | true | "no" | MSG_VALID
40             "@ Should work" | "accepted" | true | "no" | MSG_VALID
41             "1 Should work" | "accepted" | true | "no" | MSG_VALID
42             new String(maxLength) | "accepted" | true | "no" | MSG_VALID
43             "aa" | "accepted" | true | "no" | MSG_VALID
44             "Valid special chars ÅÄÅÖ0_-@./\\()&\\'+1" | "accepted" | true | "no" | MSG_VALID
45             null | "rejected" | false | "one" | MSG_INVALID_LENGTH
46             "" | "rejected" | false | "one" | MSG_INVALID_LENGTH
47             "a" | "rejected" | false | "one" | MSG_INVALID_LENGTH
48             new String(tooLong) | "rejected" | false | "one" | MSG_INVALID_LENGTH
49             "Invalid character!" | "rejected" | false | "one" | MSG_INVALID_CHAR
50             "a ok but ^ is invalid character" | "rejected" | false | "one" | MSG_INVALID_CHAR
51             "@ ok but ? is invalid character" | "rejected" | false | "one" | MSG_INVALID_CHAR
52             "- is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
53             ". is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
54             "\\ is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
55             "/" is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
56             "(" is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
57             ")" is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
58             "& is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
59             "\" is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
60             "+ is invalid start character" | "rejected" | false | "one" | MSG_INVALID_FIRST_CHAR
61     }
62 }

```

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

	NameValidatorSpec (fi.vincit.validation)	
OK	validating name 'Valid name' should be accepted as valid	68ms
OK	validating name '@ Should work' should be accepted as valid	1ms
OK	validating name '1 Should work' should be accepted as valid	0ms
OK	validating name 'aa'	1ms
OK	validating name 'aa' should be accepted as valid	0ms
OK	validating name 'Valid special chars ÅÄÅöÖ_~@./\()&+1' should be accepted as valid	2ms
OK	validating name 'null' should be rejected as valid	2ms
OK	validating name '' should be rejected as valid	2ms
OK	validating name 'a' should be rejected as valid	1ms
OK	validating name 'aa'	3ms
OK	validating name 'Invalid character!' should be rejected as valid	0ms
OK	validating name 'a ok but ^ is invalid character' should be rejected as valid	1ms
OK	validating name '@ ok but ? is invalid character' should be rejected as valid	0ms
OK	validating name '- is invalid start character' should be rejected as valid	4ms
OK	validating name '. is invalid start character' should be rejected as valid	0ms
OK	validating name '\' is invalid start character' should be rejected as valid	6ms
OK	validating name '/' is invalid start character' should be rejected as valid	0ms
OK	validating name '(' is invalid start character' should be rejected as valid	0ms
OK	validating name ')' is invalid start character' should be rejected as valid	1ms
OK	validating name '&' is invalid start character' should be rejected as valid	8ms
OK	validating name '' is invalid start character' should be rejected as valid	8ms
OK	validating name '+' is invalid start character' should be rejected as valid	0ms

Figure 5.10: Spock DDT feature method run output

Chapter 6

Results

This chapter covers the results and analysis of the case study. First the results of interview for demographic purposes are presented. Second, survey results regarding JUnit compared to Spock and Spectrum are analyzed. Third, the participant interview about BDD testing framework benefits and drawbacks are discussed. After this, the test code analysis is presented and finally with all the above mentioned results, Spock and Spectrum are compared against each other.

First interview: Demographics and projects

Surveys

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%			
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 (percentage)?	50%	20%	15%	28.33%			
3. How much of your initial effort goes to initial test case structuring and implementation (percentage)?	50%	80%	85%	71.67%			
4. How much of your overall testing effort goes to refactoring test code (percentage)?	50%	5%	30%	28.33%			
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	0	0	0	ABC	0	0	0
2. Capturing all behavior of unit/feature with tests or assertions	0	0	B	A	0	C	0
3. Execution speed	0	C	0	0	B	A	0
4. Robustness against code changes (i.e., test does not break easily)	0	0	0	0	ABC	0	0
5. How realistic the test scenario is	0	0	AB	0	0	C	0
6. How easily faults can be localised/debugged if the test fails	0	0	B	0	AC	0	0
7. How easily the test can be updated when the underlying code changes	0	C	A	0	B	0	0
8. Sensitivity against code changes (i.e., test should detect even small code changes)	0	AB	0	C	0	0	0
Q4: How difficult is it for you to understand a low level test?	Very easy	Easy	Moderate	Hard	Very hard		
	0	0	BC	A	0		
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?	0	A	0	0	BC	0	0
2. Structure and write information to stimulus of test?	0	0	0	ABC	0	0	0
3. Structure and write information to assertions of test?	0	B	C	0	A	0	0
4. Read test case structure for information about context of test?	0	0	C	0	B	A	0
5. Read test case structure for information about stimulus of test?	0	0	C	B	A	0	0
6. Read test case structure for information about assertions of test?	0	B	C	0	0	A	0
Q6: How informative you usually find the test case output?	Not at all	Hardly informative	Slightly informative	Somewhat informative	Moderately informative	Very informative	Extremely informative
	0	A	0	C	B	A	0
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)	0	0	0	0	AB	C	0
2. Lifecycle hooks Before/After -class	0	0	0	C	AB	0	0
3. Lifecycle hooks Before/After (each)	0	0	0	0	AB	C	0
4. Automatic test case generation via test case parametrization	0	A	B	C	0	0	0
5. Common test initializer class inheritance	0	AC	B	0	0	0	0
Q8: How often do you add/write documentation comments to low level test cases?	Never	Rarely	Sometimes	Fairly often	Always		
	0	C	0	AB	0		
Q9: When you make changes to low level tests, how often do you comment the changes (or update existing comments)?	Never	Rarely	Sometimes	Fairly often	Always		
	0	C	0	AB	0		
Q10: In unit testing, how many							
1. Test methods do you usually write per class method?	0	1	2-3	4-5	6-7	8-9	10 or more
2. Assertions do you usually write per test method?	0	0	AB	C	0	0	0
	0	0	AB	C	0	0	0
Q11: In unit testing, what mocking library do you normally use?	Mockito ABC	jMock 0	Powermock 0	Easymock 0	Other 0		
Q12: In unit testing, how difficult you find it to	Very easy	Easy	Slightly easy	Moderate	Slightly hard	Hard	Very hard
1. Mock objects?	0	BC	A	0	0	0	0
2. Stub method calls?	0	0	AC	0	B	0	0
3. Verify mock object actions?	0	BC	0	A	0	0	0
Q13: When do you add automated unit tests for developed code?	Before implemen- tation 0	During implemen- tation BC	After implemen- tation A				

Table 6.1: JUnit developer low level testing practice questions with response data

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	0	0	C	0	AB	0	0
2. I enjoy writing low level tests	0	B	0	A	0	C	0
3. I would like to have more tool support when writing low level tests	0	0	B	A	C	0	0
4. I would like to have more low level tests	0	0	0	ABC	0	0	0
5. Maintaining low level tests is difficult	0	0	C	0	B	A	0
6. I think my low level tests will help other developers to understand the implemented unit/feature better	0	0	B	0	A	C	0
7. Low level automated testing helps me find defects in the code before other quality assurance phases	0	0	0	0	B	C	A
8. JUnit promotes me to write high quality test code	0	0	AC	B	0	0	0
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	0	0	0	B	AC		
2. Maintaining good low level test cases and their documentations is important to the quality of a system	0	0	0	AB	C		

Table 6.2: Likert scale questions with response data about 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?	0	0	0	0	0	0	0	0	B	0	AC
2. Recommend testing framework JUnit for future Spring projects where you take part in existing project?	0	0	0	0	0	0	0	C	0	B	A
3. Take testing framework JUnit in use for future Spring projects where you have technical lead role in a new starting project?	0	0	0	0	0	0	0	0	C	B	A

Table 6.3: NPS questions with response data about developer loyalty towards low level automated testing with JUnit

Second interview: BDD framework feedback

Test code analysis

Comparison of BDD testing frameworks

Chapter 7

Discussion

At this point, you will have some insightful thoughts on your implementation and you may have ideas on what could be done in the future. This chapter is a good place to discuss your thesis as a whole and to show your professor that you have really understood some non-trivial aspects of the methods you used...

Chapter 8

Conclusions

Time to wrap it up! Write down the most important findings from your work. Like the introduction, this chapter is not very long. Two to four pages might be a good limit.

Bibliography

- [1] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification: Java SE 8 Edition*. Oracle America, 2015.
- [2] 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].
- [3] 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.
- [4] F. Buckley, “A standard for software quality assurance plans,” *Computer*, vol. 12, no. 8, 1978.
- [5] B. Kitchenham and S. L. Pfleeger, “Software quality: the elusive target [special issues section],” *IEEE Software*, vol. 13, pp. 12–21, Jan 1996.
- [6] P. J. Denning, “Software quality,” *Commun. ACM*, vol. 59, pp. 23–25, Aug 2016.
- [7] D. Chelmsky, 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.
- [8] 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.
- [9] N. Cerpa and J. M. Verner, “Why did your project fail?,” *Communications of the ACM*, vol. 52, no. 12, pp. 130–134, 2009.
- [10] 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].

- [11] 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.
- [12] 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.
- [13] 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.
- [14] J. Itkonen, “Lecture: Building quality in modern software development,” October 2016.
- [15] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.
- [16] R. Osherove, *The Art of Unit Testing, Second Edition*. Manning Publications Company, 2013.
- [17] P. Runeson, “A survey of unit testing practices,” *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.
- [18] J. A. Whittaker, “What is software testing? and why is it so hard?,” *IEEE software*, vol. 17, no. 1, pp. 70–79, 2000.
- [19] J. Langr, A. Hunt, and D. Thomas, *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf, 2015.
- [20] K. Kapelonis, *Java Testing with Spock*. Manning Publications Company, 2016.
- [21] 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.
- [22] 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.
- [23] L. Williams, G. Kudrjavets, and N. Nagappan, “On the effectiveness of unit test automation at microsoft.,” in *ISSRE*, pp. 81–89, 2009.

- [24] 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.
- [25] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [26] 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.
- [27] K. Beck, “Aim, fire [test-first coding],” *IEEE Software*, vol. 18, no. 5, pp. 87–89, 2001.
- [28] 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.
- [29] 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.
- [30] D. Astels, “A new look at test-driven development,” http://blog.daveastels.com/files/BDD_Intro.pdf, Accessed em, vol. 12, p. 2013, 2006.
- [31] E. Amodeo, *Learning Behavior-driven Development with JavaScript*. Community Experience Distilled, Packt Publishing, 2015.
- [32] 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.
- [33] M. Gärtner, *ATDD by example: a practical guide to acceptance test-driven development*. Addison-Wesley, 2012.
- [34] 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.
- [35] DevelopSense, “Blog: Acceptance tests: Let’s change the title, too.” <http://www.developsense.com/blog/2010/08/acceptance-tests-lets-change-the-title-too/>, 2010. [Online; accessed 24-March-2017].

- [36] 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.
- [37] D. North, “Introducing bdd.” <https://dannorth.net/introducing-bdd/>, 2006. [Online; accessed 24-March-2017].
- [38] 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.
- [39] J. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications Company, 2014.
- [40] 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.
- [41] R. M. Lerner, “At the forge: Rspec,” *Linux Journal*, vol. 2009, no. 186, 2009.
- [42] 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].
- [43] 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.
- [44] 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.
- [45] L. Chantal and S. Regina, “Towards and empirical evaluation of behavior-driven development,” 2009.
- [46] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development*. ” O’Reilly Media, Inc.”, 2004.
- [47] JUnit, “JUnit - about.” <http://junit.org/junit4/>, 2017. [Online; accessed 22-March-2017].
- [48] Github, “Roadmap Â· junit-team/junit5 wiki Â· github.” <https://github.com/junit-team/junit5/wiki/Roadmap>, 2017. [Online; accessed 22-March-2017].

- [49] P. Software, “Spring framework.” <https://projects.spring.io/spring-framework/>, 2017. [Online; accessed 22-March-2017].
- [50] 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].
- [51] 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].
- [52] G. C. Archive, “The hamcrest tutorial.” <https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki>, 2017. [Online; accessed 04-April-2017].
- [53] G. Inc, “Gradle build tool.” <https://gradle.org/>, 2017. [Online; accessed 30-March-2017].
- [54] Artima, “Scalatest - getting started with funspec.” http://www.scalatest.org/getting_started_with_fun_spec, 2017. [Online; accessed 27-March-2017].
- [55] RSpec, “RSpec documentation.” <http://rspec.info/documentation/>, 2017. [Online; accessed 28-March-2017].
- [56] RSpec, “rspec-core.” <http://rspec.info/documentation/3.5/rspec-core/>, 2017. [Online; accessed 28-March-2017].
- [57] Mockito, “Mockito framework site.” <http://site.mockito.org/>, 2017. [Online; accessed 30-March-2017].
- [58] 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].
- [59] JetBrains, “IntelliJ idea the java ide.” <https://www.jetbrains.com/idea/>, 2017. [Online; accessed 30-March-2017].
- [60] RubyGems, “What is a gem? - rubygems guides.” <http://guides.rubygems.org/what-is-a-gem/>, 2017. [Online; accessed 30-March-2017].
- [61] 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].

- [62] 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].
- [63] Github, “Gherkin.” <https://github.com/cucumber/cucumber/wiki/Gherkin>, 2017. [Online; accessed 28-March-2017].
- [64] Github, “jimweirich/rspec-given: Given/when/then keywords for rspec specifications.” <https://github.com/jimweirich/rspec-given>, 2017. [Online; accessed 28-March-2017].
- [65] Artima, “Scalatest - getting started with featurespec.” http://www.scalatest.org/getting_started_with_feature_spec, 2017. [Online; accessed 28-March-2017].
- [66] Github, “Pease.” <http://pease.github.io/>, 2017. [Online; accessed 28-March-2017].
- [67] 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.
- [68] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.
- [69] 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].
- [70] LimeSurvey, “Limesurvey: the online survey tool - open source surveys.” <https://www.limesurvey.org/>, 2017. [Online; accessed 06-April-2017].
- [71] 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.
- [72] F. F. Reichheld, “The one number you need to grow,” *Harvard business review*, vol. 81, no. 12, pp. 46–55, 2003.
- [73] Eclemma, “Eclemma jacoco java code coverage library.” <http://www.eclemma.org/jacoco/>, 2017. [Online; accessed 07-April-2017].

- [74] M. G. . C. KG and Contributors, “Jacoco coverage counter.” <http://www.jacoco.org/jacoco/trunk/doc/counters.html>, 2017. [Online; accessed 07-April-2017].
- [75] Wikipedia, “Camel case — wikipedia, the free encyclopedia,” 2017. [Online; accessed 07-April-2017].
- [76] Github, “Parameterized tests Â· junit-team/junit4 wiki,” 2017. [Online; accessed 12-April-2017].

Appendix A

Gradle build configurations

```
35 configurations {
36     integrationTestCompile.extendsFrom testCompile
37     integrationTestRuntime.extendsFrom testRuntime
38     acceptanceTestCompile.extendsFrom testCompile
39     acceptanceTestRuntime.extendsFrom testRuntime
40 }
41 dependencies {
42     testCompile 'junit:junit:' + junitVersion
43     testCompile 'org.hamcrest:hamcrest-all:' + hamcrestVersion // Hamcrest matchers
44     testCompile 'org.mockito:mockito-core:' + mockitoVersion //Mockito for mocking/stubbing
45     testCompile 'io.rest-assured:rest-assured:' + restAssuredVersion //REST-integration testing
46     //Spring domain extension
47     integrationTestCompile 'org.springframework.boot:spring-boot-starter-test'
48 }
49 sourceSets {
50     integrationTest {
51         java {
52             compileClasspath += main.output + test.output
53             runtimeClasspath += main.output + test.output
54             srcDir file('src/integration-test/java')
55         }
56         resources.srcDir file('src/integration-test/resources')
57     }
58 }
59 task integrationTest(type: Test) {
60     testClassesDir = sourceSets.integrationTest.output.classesDir
61     classpath = sourceSets.integrationTest.runtimeClasspath
62     outputs.upToDateWhen { false }
63 }
64 check.dependsOn test
65 check.dependsOn integrationTest
66 integrationTest.mustRunAfter test
```

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

```
55 configurations {
56     integrationTestCompile.extendsFrom testCompile
57     integrationTestRuntime.extendsFrom testRuntime
58     acceptanceTestCompile.extendsFrom testCompile
59     acceptanceTestRuntime.extendsFrom testRuntime
60 }
61 dependencies {
62     testCompile 'junit:junit:' + junitVersion
63     testCompile 'org.hamcrest:hamcrest-all:' + hamcrestVersion // Hamcrest matchers
64     testCompile 'org.mockito:mockito-core:' + mockitoVersion //Mockito for mocking/stubbing
65     testCompile 'io.rest-assured:rest-assured:' + restAssuredVersion //REST-integration testing
66     testCompile 'com.greghaskins:spectrum:' + spectrumVersion //Spectrum JUnit runner
67     //Spring domain extension
68     integrationTestCompile 'org.springframework.boot:spring-boot-starter-test'
69 }
70 sourceSets {
71     integrationTest {
72         java {
73             compileClasspath += main.output + test.output
74             runtimeClasspath += main.output + test.output
75             srcDir file('src/integration-test/java')
76         }
77
78         resources.srcDir file('src/integration-test/resources')
79     }
80 }
81 task integrationTest(type: Test) {
82     testClassesDir = sourceSets.integrationTest.output.classesDir
83     classpath = sourceSets.integrationTest.runtimeClasspath
84     outputs.upToDateWhen { false }
85 }
86 check.dependsOn test
87 check.dependsOn integrationTest
88 integrationTest.mustRunAfter test
```

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

```
54 configurations {
55     integrationTestCompile.extendsFrom testCompile
56     integrationTestRuntime.extendsFrom testRuntime
57     acceptanceTestCompile.extendsFrom testCompile
58     acceptanceTestRuntime.extendsFrom testRuntime
59 }
60 dependencies {
61     testCompile 'org.codehaus.groovy.modules.http-builder:http-builder:0.6' //Rest-client
62     testCompile 'org.codehaus.groovy:groovy-all:2.4.4'
63     testCompile 'org.spockframework:spock-core:1.1-groovy-2.4-rc-3' //Spock
64     testCompile 'cglib:cglib-nodep:3.2.4' //stubbing classes
65
66     testCompile ('com.athaydes:spock-reports:1.2.13') { //BDD-reports
67         transitive = false // this avoids affecting your version of Groovy/Spock
68     }
69     //InjectMocks type mocking (not recommended)
70     testCompile 'com.blogspot.toomuchcoding:spock-subjects-collaborators-extension:1.2.1'
71     integrationTestCompile 'org.spockframework:spock-spring:1.1-groovy-2.4-rc-3' //Spock-spring
72 }
73 sourceSets {
74     integrationTest {
75         groovy {
76             compileClasspath += main.output + test.output
77             runtimeClasspath += main.output + test.output
78             srcDir file('src/integration-test/groovy')
79         }
80         resources.srcDir file('src/integration-test/resources')
81     }
82 }
83 task integrationTest(type: Test) {
84     testClassesDir = sourceSets.integrationTest.output.classesDir
85     classpath = sourceSets.integrationTest.runtimeClasspath
86     include '**/*Spec.*'
87     outputs.upToDateWhen { false }
88 }
89 check.dependsOn test
90 check.dependsOn integrationTest
91 integrationTest.mustRunAfter test
```

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


```

155 configurations { rspec }
156 dependencies {
157     rspec 'org.jruby:jruby-complete:' + jRubyVersion
158 }
159 task(bundler, type: JavaExec) {
160     main = 'org.jruby.Main'
161     classpath = configurations.rspec
162     args = ['-S', 'gem', 'install', 'bundler']
163     environment['GEM_PATH'] = file('build/gems').path
164     environment['GEM_HOME'] = file('build/gems').path
165 }
166 task(gems, dependsOn: ["bundler"], type: JavaExec) {
167     main = 'org.jruby.Main'
168     classpath = configurations.rspec
169     args = ['-S', 'bundle', 'install']
170     environment['GEM_PATH'] = file('build/gems').path
171     environment['GEM_HOME'] = file('build/gems').path
172 }
173 task(spec, dependsOn: ["classes"], type: JavaExec) {
174     main = 'org.jruby.Main'
175     classpath = sourceSets.test.runtimeClasspath + configurations.rspec
176     if ( project.hasProperty("file") ) {
177         args = ['-S', 'build/gems/bin/rspec', 'src/spec/unit/' + file]
178     } else {
179         args = ['-S', 'build/gems/bin/rspec', 'src/spec/unit']
180     }
181     environment['GEM_HOME'] = file('build/gems').path
182     environment['GEM_PATH'] = file('build/gems').path
183     environment['ENV'] = "test"
184 }
185 task(integrationSpec, dependsOn: ["classes"], type: JavaExec) {
186     main = 'org.jruby.Main'
187     classpath = sourceSets.test.runtimeClasspath + configurations.rspec
188     if ( project.hasProperty("file") ) {
189         args = ['-S', 'build/gems/bin/rspec', 'src/spec/api/' + file]
190     } else {
191         args = ['-S', 'build/gems/bin/rspec', 'src/spec/api']
192     }
193     environment['GEM_HOME'] = file('build/gems').path
194     environment['GEM_PATH'] = file('build/gems').path
195     environment['ENV'] = "integration-test"
196 }
197 check.dependsOn spec
198 check.dependsOn integrationSpec
199 integrationSpec.mustRunAfter spec

```

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

Appendix B

Interview questions

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?
6. What are your expectations regarding the switch to a new testing framework?

Appendix C

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.

Survey regarding JUnit in automated low level testing

Survey regarding JUnit contains base questions from multiple previous research studies [23] [43] [44]. 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 research:

- **Q11, Q12:** Developers finding isolating of unit under test hard []
- **Q14:** Only half of the survey respondents enjoy writing unit tests []
- **Q2, Q7, Q14, Q15:** Maintaining unit tests was found harder than writing them []
- **Q4, Q5, Q6:** For 60.38% of developers, understanding unit tests is at least moderately difficult []
- **Q8, Q9:** Developers find updated documentation and comments in test cases useful, but writing comments to unit tests is rarely or never done []

- **Q14, Q15:** Majority of developers find unit tests helpful in producing higher quality code []
- **Q14:** Majority of developers find unit tests helpful in understanding other peoples code []
- **Q15:** Neglecting test case maintenance effort can result in test cases that lose their information value running capability []
-

Table C.1 displays all the questions in the JUnit survey. It contains 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?* [43]. Q3 Subquestions 1, 3-8 are originals. Q3 Subquestion 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?* [44]. 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)?* [44].

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 case 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 case generation via test case 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 C.1: 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 C.2: 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 C.3: NPS questions of developer loyalty towards low level automated testing with JUnit

Survey regarding BDD testing framework in automated low level testing

Question	Answer options											
Q1: Compared to JUnit, How do you spend your software development time?	A lot less time	Less time	Slightly less time	Less	The same amount	Slightly more time	More time	A lot more time				
1. Writing new code 2. Writing new tests 3. Debugging/fixing 4. Refactoring 5. Other												
Q2: Compared to JUnit, How do you spend your low level automated testing time?	A lot less	Less	Slightly less	Less	The same amount	Slightly more	More	A lot more				
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?												
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	Less	As important as before	Slightly more important	More important	A lot more 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: Compared to JUnit, how difficult is it for you to understand a low level test?	A lot less difficult	Less difficult	Slightly difficult	Less	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.	A lot less difficult	Less difficult	Slightly difficult	Less	As difficult as before	Slightly more difficult	More difficult	A lot more difficult				
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: Compared to JUnit, how informative you usually find the test case output?	A lot less informative	Less informative	Slightly less informative	Less	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?	A lot less	Less	Slightly less	Less	The same amount	Slightly more	More	A lot more				
1. Extract method (custom helper methods) 2. Lifecycle hooks Before/After -class 3. Lifecycle hooks Before/After (each) 4. Automatic test case generation via test case parametrization 5. Common test initializer class inheritance												
Q8: Compared to JUnit, how often do you add/write documentation comments to low level test cases?	A lot less	Less	Slightly less	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	Less	The same amount	Slightly more	More	A lot more				
Q10: Compared to JUnit in unit testing	A lot less	Less	Slightly less	Less	The same amount	Slightly more	More	A lot more				
1. Do you write more or less test methods per class method? 2. Do you write more or less assertions per test method?												
Q11: In unit testing with <i>Spectrum/Spock</i> , what mocking library do you normally use?	<i>Spock's internal mocking</i>	Mockito	jMock		Powermock	Easymock	Other					
Q12: Compared to JUnit in unit testing, how difficult you find it to	A lot easier	Easier	Slightly easier	Easier	As difficult as before	Slightly harder	Harder	A lot harder				
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 C.4: *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 <i>Spectrum/Spock</i> is more difficult than with JUnit							
2. I enjoy writing low level tests with <i>Spectrum/Spock</i> more than I do with JUnit							
3. I would like to have more tool support for <i>Spectrum/Spock</i> when writing low level tests							
4. I would like to have more low level tests for <i>Spectrum/Spock</i>							
5. Maintaining low level tests with <i>Spectrum/Spock</i> is more difficult than with JUnit							
6. I think my low level tests with <i>Spectrum/Spock</i> will help other developers to understand the implemented unit/feature better than earlier tests with JUnit							
7. Low level automated testing with <i>Spectrum/Spock</i> helps me find defects in the code before other quality assurance phases better than earlier tests with JUnit							
8. <i>Spectrum/Spock</i> promotes me to write higher quality test code than with JUnit							
9. Overall, low level tests with <i>Spectrum/Spock</i> help me produce higher quality code than with JUnit							
10. Maintaining good low level test cases and their documentation with <i>Spectrum/Spock</i> is more important for system quality than maintaining JUnit test cases							

Table C.5: Likert scale questions of developer perception towards *Spectrum/Spock*

Question	Answer options										
Q15: 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 <i>Spectrum/Spock</i> for future Spring projects where you take part in existing project?											
3. Take testing framework <i>Spectrum/Spock</i> in use for future Spring projects where you have technical lead role in a new starting project?											

Table C.6: NPS questions of developer loyalty towards low level automated testing with *Spectrum/Spock*

Question	Answer options		
Q16: I would say that I write more	Yes	Uncertain	No
1. Understandable low level tests with <i>Spectrum/Spock</i> than with JUnit?			
2. Maintainable low level tests with <i>Spectrum/Spock</i> than with JUnit?			

Table C.7: Questions of developer perception towards *Spectrum/Spock*