Fault detection power of unit and system testing in Java open source projects

Master's thesis in Computer science and engineering

ANDREEA SULUGIU

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Fault detection power of unit and system testing in Java open source projects

ANDREEA SULUGIU

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Fault detection power of unit and system testing in Java open source projects

ANDREEA SULUGIU

Supervisor: Robert Feldt, Department of Computer Science and Engineering
Examiner: Gregory Gay, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2021

Fault detection power of unit and system testing in Java open source projects

ANDREEA SULUGIU
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Testing is one of the most crucial steps in providing quality for software products. Two key and heavily used testing levels are unit and system testing, each level having different benefits and drawbacks. For their testing process a software team needs to decide where the focus and effort should be put, often by creating a strategy for testing. Because the resources are limited, a critical question is “what is a good trade-off between different levels of testing in order to maximize the effect, i.e. quality improvement and assurance while using as few resources as possible”. There are a lot of factors to consider for this trade-off, from costs to time and knowledge of testers, but in this study we focus on a critical one: the fault-finding ability and behaviour of tests on different levels.

To evaluate the fault finding behaviour of unit and system tests, a reliable method for test level categorization is needed. Based on the attributes found in the common definitions for the testing levels, a framework for categorization was developed and applied to analyse the usage of unit and system level testing on selected Java open source projects from the Defects4j framework. Furthermore, using information provided by the Defects4j tool the fault detection level of different tests and, ultimately, of the different levels can be determined. The 16 analysed projects contained 25477 tests, where 78.4% of tests were categorized as low level and 21.56% as high level. The results indicate that lower, unit level tests are used more in the investigated Java open source projects. Looking at fault finding ability, from the 25477 tests, only 998 tests were able to uncover bugs. From these 998 tests, 65.73% of tests were categorized as low level and 34.27% were categorized as high level. It should be mentioned that this result can be attributed to the fact that there was a higher number of low level tests in the sample. Considering the rate of bug discoverability from the total number of tests on the respective levels, the high level performed better, with a discoverability rate of 6.23% while the low level had a rate of 3.28%.

The major contribution of this paper is a framework for test categorization on a low- to a high-level scale, based on concrete and objective metrics that can be practically applied on projects that subscribe to the object-oriented paradigm. Additionally, backed on empirical data, we found out that high level tests have a higher rate of discovering bugs.

Keywords: Software testing, unit test, system test, test categorization, bugs, project, thesis, fault detection.

Acknowledgements

I would like to thank my supervisor Robert Feldt for his support, guidance and feedback throughout the project. I wish to acknowledge the support of my husband, who provided me with professional input and continuous encouragement through the process of researching and writing this thesis.

Andreea Sulugiu, Gothenburg, June 2021

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
- 1.1 Introduction . . . 1
- 1.2 Statement of the problem . . . 3
- 1.3 Statement of the purpose . . . 4
- 1.4 Research questions . . . 4

2 Background and Related Work **7**
- 2.1 Definitions for testing levels . . . 7
- 2.2 Test type classification . . . 8
- 2.3 Testing in open source projects . . . 9
- 2.4 Comparison between levels of testing . . . 9

3 Study Design **11**
- 3.1 Data collection . . . 11
  - 3.1.1 Defects4J data collection . . . 11
  - 3.1.2 Test data collection . . . 12
  - 3.1.3 Data exclusion constraints . . . 13
  - 3.1.4 Sample . . . 14
- 3.2 Analysis Process . . . 15

4 Framework **17**
- 4.1 Definitions and Characteristics . . . 17
- 4.2 Metrics . . . 20
  - 4.2.1 Approach for size . . . 21
  - 4.2.2 Approach for isolation . . . 24
- 4.3 Test score and Scale . . . 25

5 Results **31**
- 5.1 RQ1 . . . 31
- 5.2 RQ2 . . . 34
- 5.3 RQ3 . . . 39
- 5.4 Projects Information . . . 55

6 Discussion **58**
6.1 Limitations 60
6.2 Implications 61
6.3 Validity Consideration 62

7 Conclusion **64**
7.1 Conclusion 64
7.2 Future work 64

Bibliography **66**

A Appendix 1 **I**
A.1 RQ tables I

List of Figures

5.1 Proportion of low and high level tests per project 35
5.2 Number of low and high level tests per project 36
5.3 a) Low and high level test percentage vs b) Pure unit test percentage 37
5.4 Density of test scores 38
5.5 Histogram of test scores 38
5.6 Score distribution per project 39
5.7 Proportion of low and high level trigger tests 42
5.8 Number of low and high level trigger tests 43
5.9 Density of test scores 43
5.10 Histogram of test scores 44
5.11 Score density per project 44
5.12 Low-High tests difference for bugs 46
5.13 Low-High tests difference for bugs 46
5.14 Relationship between no. of methods in test and bugs uncovered . . 47
5.15 Relationship between no. of methods in test, bugs uncovered and testing level 48
5.16 Relationship between no. of methods in code tested and bugs uncovered 48
5.17 Relationship between no. of methods in code tested, bugs uncovered and testing level 49
5.18 Relationship between no. of classes in test and bugs uncovered 49
5.19 Relationship between no. of classes in test, bugs uncovered and test level 50
5.20 Relationship between no. of classes in code tested and bugs uncovered 50
5.21 Relationship between no. of classes in code tested, bugs uncovered and test level 51
5.22 Relationship between no. of packages in test and bugs uncovered . . 52
5.23 Relationship between no. of packages in test, bugs uncovered and test level 52
5.24 Relationship between no. of packages in code tested and bugs uncovered 53
5.25 Relationship between no. of packages in code tested, bugs uncovered and test level 53
5.26 Relationship between big setup and bugs uncovered 54
5.27 Relationship between big setup, bugs uncovered and level of test . . . 54
5.28 Setup state per project 57

List of Tables

3.1 Defects4J projects [50] 14

4.1 ISTQB definitions [9] 18
4.2 IEEE 24765:2017 standard definitions [8] 18
4.3 Attributes based on definition 20
4.4 Discussed metrics for size 24
4.5 Characteristics and metrics 25
4.6 Test scores for size of test 27
4.7 Test scores for size of code tested 27
4.8 Test scores for size of setup 28
4.9 Test scores for isolation 28

5.1 Characteristics and metrics 31
5.2 Projects bug information 40
5.3 Projects size regarding tests 56
5.4 Mock usage 56

A.1 Projects version I
A.2 Framework verification II
A.3 Second framework verification III
A.4 Second framework verification IV
A.5 Second framework verification V
A.6 Second framework verification VI
A.7 Test categorization per project VI
A.8 Triggered tests categorization per project VII
A.9 Setup usage per project VII
A.10 Big setup usage per project VIII

1 Introduction

1.1 Introduction

Software testing is an activity performed in order to assess the quality of a piece of software. This is important both in finding and fixing any remaining bugs but also in judging when a system is stable enough and thus can be deployed to end users. There are different methods and testing techniques as well as different levels of testing (unit, integration and system) each with their advantages and disadvantages.

Unit tests are fast, as they only verify the functionality of a small unit of code, thus the feedback from the test is fast and can be easily traced to the problem. They can be used early in the development cycle, which is an advantage because discovering bugs later is more costly. A problem with unit tests comes from the sheer number of tests that should be done to cover all the units in a system. A comprehensive set of unit tests will verify the units in separation of the other parts of the program.

Based on the 14th Annual State of Agile Report [1], 95% of the organizations included in the survey practice some form of Agile methods. From the Agile practices, the most popular one is Scrum, with 58%, while the next 27% practice a hybrid form between Agile methods. From the engineering practices employed by the organization in the spirit of being Agile, unit testing is the first one with a percentage of 67% while Test Driven Development is at 30%. With the rise of Agile practices, unit testing became a staple of software development processes. Although in the Agile manifesto unit testing or any other testing level or method is not mentioned, technical excellence, which is closer to lower-level, unit tests, is in focus. From the different Agile practices, unit testing is integral to at least two of them: Extreme Programming through Test-Driven Development and Scrum.

One of the four main activities in the Extreme Programming agile method is testing, with unit testing being the main effort in determining if a feature works correctly. This is then complemented by higher-level, acceptance tests to determine if the requirement was correctly understood and if it satisfies the client's need. More internally focused system tests are used less often. The practice of Test-Driven Development relies on creating tests before developing specific software methods or functions, and is thus typically focused on the unit testing level. Test-Driven Development is the embodiment of test-first programming concept found in Extreme programming, but can be practiced on its own [2]. And in Scrum, its Definition of Done criterion (a criterion for a user story / feature to be considered completed) usually includes unit testing.

Unit testing and acceptance testing are integral elements of agile methodologies, as well as automated tests. In literature, there is no need to mention automated tests when discussing testing in agile as it is implied, in the same way that unit tests don't need to be mentioned as they are assumed as such, if not stated otherwise [3]. Unit testing as a concept applies perfectly for the need of testing in each iteration, which can be used from the start of projects. Unit testing is a daily occurrence in agile software development, while system testing and acceptance testing are done at the end of an iteration, before deploying the product in production. This suggests a bottom-up approach in agile testing strategies. Brian Marick introduced the concept of the Agile Testing Quadrant in 2003, which has been updated in consequent years. In all versions of the matrix unit testing is present in the first quadrant, which is focused on technology, guiding development and preventive testing. System testing can be found between the second and third quadrant, while acceptance testing is present in the third quadrant, which is more focused on the business part and detection testing. Because of the prevalence of the unit testing technique, it would be considered as the primary testing technique, comparing its fault finding behaviour with that of system testing.

System tests are conducted on the entire system, thus they more often represent realistic uses of the system and they check the requirements. Because system testing is conducted on the entire system, this approach cannot be done early in the development cycle, as the full system would evolve and the previous system tests would become obsolete. Thus, system testing is mainly done late in the development cycle, so the cost of fixing bugs found at that time is higher. A drawback of system testing is the fact that there is a big distance between the failure and the fault, so it is harder to find the location of the faulty code from the system test.

Since development and testing resources are not infinite, it's important for practitioners to know the trade-off between the different levels of testing, to be able to determine which approach is better for their situation and needs. There are a lot of things to consider when talking about the trade-off, from the detection power of the method, to cost and time of execution, types of bugs detected, expertise needed, return on investment (ROI) etc.

The cost of testing is different based on the level of testing. Unit testing can be done in the first stages of development, preventing bugs to advance through the life cycle of the project, where it would be expensive to fix them [4]. The system testing is done late in the development stages, being the last barrier before acceptance testing. The bugs found at this level are costly. However, unit testing is typically a white-box testing technique, so the people that write the test must have the detailed knowledge of the system they are testing, regardless if they are dedicated testers or the developer that wrote the code [5]. On the contrary, system testing is a black-box technique, where the testers which are independent from the development team need not have the same, detailed knowledge of the inner working of the system [5].

For unit testing, the feedback is fast and the fault can often be more quickly identified, as the test checks smaller units of code. On the other side, the feedback time for a system test is longer and it takes considerable effort to find the fault, as the

test includes the whole system.
While information about the different testing levels is documented thoroughly, there is no clear analysis of the trade-off of software quality and fault detection between unit testing and system testing. There is a need for empirical data to decide which testing level can produce a more comprehensive fault detection coverage, in order to increase the quality of the software.

1.2 Statement of the problem

Fault finding behaviour of testing levels
In an ideal scenario a software development project would have limitless resources at its disposal, which would lead to a fully comprehensive testing suite on all the different levels of testing. In a real world scenario, though, a project is bound by a varying set of factors such as cost, time and human resources. This leads to a higher effort in managing testing throughout a project's life-cycle, as testing efforts are costly in time and money. Because of the limited nature of the aforementioned resources, the software team has to decide where the testing should be concentrated, which testing methods they will apply and prioritize how much of each method they will use, in order to maximize the outcome. Research has shown that typically between 40% to 70% of the time and cost of a software project can be attributed to software testing [6]. However, very little is known about how to assign this effort between different testing activities.
In industry, the "testing pyramid" is sometimes used to help guide in making the decision of how much of each level of testing should be applied for a project [7], but this is not supported by empirical data that links the levels to actual project costs and benefits. Unit tests are fast and easy to write but they lack software context, while system tests are slow and harder to write but are both more realistic as well as closer to users' needs. So, given bounded resources for testing, a basic question is how to find a good trade-off between unit tests and system tests? An element to consider in this trade-off is the fault finding behaviour of the testing levels, which is our main concern in this study. We focus on maybe the two most common testing levels, unit and system testing, as unit testing is a staple of software testing and system testing is the last barrier before the product is released.

Testing level definition
Unit tests and system tests are some of the testing techniques used in industry. There are multiple definitions for the terms, two of the most popular ones come from the IEEE standard [8] and ISTQB Glossary [9]. As software development evolved, the previously mentioned definitions did not change to reflect the new status quo [10]. There are problems with what is understood as a unit in testing. A function or a procedure could be considered a unit in procedural programming, while in object-oriented programming a unit can be a class or a method [11]. The difference in the interpretation of units can vary based on paradigm, language or interpretation of developers. Because of this fact and the misuse of the terms by software developers [12], a new categorization method had to be created in order to standardize the grouping of tests.

1.3 Statement of the purpose

The purpose of the study is to evaluate what testing levels are the most used in Java open source projects, how they differ in their fault detection ability and if system testing can uncover the same bugs as unit testing through their detection power. The test cases from the testing suite would be categorized on different levels based on a developed standardized method, followed with an analysis on the fault detection behavior of the test suites. At the end of the study, we would have a fully developed framework for classification of tests on a scale from low-level to high-level that can act as a surrogate or substitute for the levels we currently have, and empirical data about the fault detection power of the substitute levels.

1.4 Research questions

The following research questions guide the research into the fault detection power of unit and system testing.

RQ1: *What metrics can act as proxy for the testing levels?* Because of inconsistencies and misuse of the terms unit and system testing, there is a need to create a framework to categorize tests between low-level and high-level. To do that we need to determine a set of metrics that can clearly differentiate the tests on the specific scale.
RQ2: *What level of testing is used more in open source projects between unit testing and system testing?* This is an investigation into the testing activities for open source projects.
RQ3: *Does unit or system testing discover more bugs, and what overlap in bug detection exists between the two techniques?* The goal is to determine which technique can discover more bugs and if the respective bugs can be found by both techniques. This research question is broken down in two parts, each part linked to a hypothesis.
Hypothesis 1: *Using system tests leads to a greater fault detection than using unit testing.* This hypothesis covers the first part of the third research question. We are interested to see what level of testing can discover more bugs and how do the detection rate compare between the two levels. We believe that system level would have a higher fault detection rate.
Hypothesis 2: System testing can discover the same bugs as unit testing. This hypothesis covers the second part of the third research question and it is concerned with how and from what level the bugs are uncovered. To make an assessment about it, we look at the bugs that were uncovered by multiple triggering tests and determine the number of bugs that were uncovered by both level of tests compared to the number of bugs uncovered by each level. We believe that the system level can uncover the same bugs as unit testing.

While RQ3 is the main focus of our work we need to first answer questions RQ1 and RQ2 to be able to then investigate RQ3 on a set of actual software projects. RQ2 focuses on open-source projects since it is not feasible in a thesis project to get access to and then investigate in depth multiple industrial projects.

2
Background and Related Work

In this chapter we present several research papers linked to our research domain. First, we make a case about the relevance of the unit and system testing terms in the current context of software development and the fact that developers might misuse or wrongly identify tests. A classification based on different definitions for the terms would yield different results hence there is a need to develop a uniform method to classify the tests. We present a number of categorization methods that other researcher developed. Moreover, we discuss the state of testing in open source projects in regards to adoption and adequacy of testing methods. Finally, we present research into the comparison of the testing levels.

2.1 Definitions for testing levels

As found by Trautsch et al [12] developers are using the term "unit testing" for their tests even when the definition for the term does not apply to the situation. The definitions used in Trautsch's study came from the IEEE [8] and ISTQB [9] and both definitions for unit and unit testing were used in the analysis. They examined 10 python projects with a total number of 4637232 tests, calculated by summing up the number of tests in each revision. The results showed that the developers' categorization of unit tests is different from the categorization based on both definitions, as consistently throughout the 10 projects analysed the number of developer unit tests was larger than the unit tests categorized based on the definitions. Moreover, there is a difference in categorization of tests based on which definition is used, as it was identified that the unit tests based on the ISTQB standard are a subset of the unit tests that are based on the IEEE definition.
Furthermore, the definition for unit testing seems to not be in line with the modern software development context [10]. The authors acknowledged that the definitions did not keep up with the advances made in the software development context and they propose based on their research a more coarse-granular definition for unit, as they identified a problem with the too fine-grained definition of unit in the current context. The proposed definition is "a test that only assesses parts that are within one component [of a complex software system]" [10].
In fact, there are multiple definitions and interpretation of the concept of unit testing, such as Whittaker's definition: "unit testing tests individual software components or a collection of components" [13] which mentions a collection of components as a unit. In Runeson's survey about unit testing practices, there were different opinions about what unit means, but in response of what unit test means the majority

strongly agreed on " a test of the smallest separate unit" and "a technical test with in/out parameters" as two criteria of unit testing [14].
We agree that the definitions for the testing terms are not representative to the current context of software testing and the fact that different definitions and developer characterizations can vary from each-other means that there is a need of an objective, measurable way to categorize test levels.

2.2 Test type classification

The difference in fault detection power between testing levels and categorizing tests was tackled by different researchers. Orellana et. al. [41] researched the difference between unit and integration tests in the TravidTorrent Dataset, where it was discovered that unit tests exposed more defects than integration tests. This result can be attributed to the fact that there were significantly more unit tests than integration tests. The strategy of classification for the tests into unit and integration was based on two heuristics, one based on the usage of Maven [43] plugins and the other based on naming conventions. The Maven build system contains two plugins for executing tests: SureFire which is used for unit tests and FailSafe which is used for integration tests. These plugins can be configured to only run tests from a source location and include files with a certain pattern. The second heuristic employs a naming convention which dictates that tests would be placed in integration level if the test name contains "IntegrationTest" or the abbreviated term "IT". The approach is based on the classification made by the developers or testers, and that can be problematic, as it was shown previously that the developers' characterization is not in line with characterization based on definitions. Moreover, this approach brings the limitation of analysing projects that use Maven as a build system and both of the plugins for test management. We deemed this approach as to subjective as it is relaying on developers for categorization. In addition, in the open source context, we fear that the naming conventions would not be upheld.
Rothermel et al. [40] based the categorization on the test case input, where a higher amount of input would translate to a higher level. The researchers defined a test case as "consisting of a pretest state of the system under test, a sequence of test inputs, and a statement of expected results" [40]. The drawback of this approach is the fact that we have no information about the size of code executed, and this can be a good source of information in the categorization process.
Another approach at categorization was taken by Kanstrén [39]. He proposed a quantitative method for measuring test coverage that takes into consideration the different testing levels, by taking into account how the different level of testing are covering the system under test. To do that, he had to rely on a quantitative method for differentiating the level of tests. He measures test granularity by counting the number of unique methods that were covered by a test, regardless of the place of invocation, then maps the granularity to the level of testing. The decision of counting all the methods reached to an end of call depth makes sense in the context of his research, as a test that executes code that ultimately executes a method would count as coverage for that specific method, but for our context the level of depth for counting the methods is too extreme.

2.3 Testing in open source projects

Studies about adoption and adequacy of testing in open source projects were done by P. S. Kochhar et al [15] [16], where they found that from 20,817 projects of different sizes, 61.65% contained test cases, Java language being the most popular language that had projects with test cases. In addition, they determined a weak positive correlation between the number of test cases from a project with the number of bugs, as well as bug reporters. Code coverage was analysed for 327 Java open source projects, where the average coverage was 41.96%, and almost a third of the projects had coverage less than 25% and over half the project had under 50% coverage. Farooq and Quadri studied the quality practices employed in open source software development, where they discovered that only high-profile open source projects employ formal and automation testing due to the need of sponsorship and expertise required [17] [18]. In open source projects undocumented and unsystematic testing process is more common.

2.4 Comparison between levels of testing

A comparison between unit testing and integration testing regarding the defect detection capabilities was done by Trautsch et. al. [10] which concluded that there were no significant differences between the two approaches. Functional testing and structural testing were compared on fault detection effectiveness and cost, where functional testing detected more faults than structural testing when professional programmers applied the testing techniques [19]. Other studies compare the effectiveness of different testing methods [20] [21].

Multiple studies were made to study the defects finding rate between methods, such as inspection vs. testing, functional testing vs. structural testing and inspections, and they were summarised in a survey of defect detection studies [24]. One such study was done by Berling et.al. [25] and it researched the difference between inspection, unit, subsystem and system tests on the following artifacts: requirements, design and code. The case study suggests that the most appropriate test effort would be unit tests and low level design. The authors proposed a measure to determine if a fault could have been found earlier in the process, and based on the measure the outcome indicates that the majority of the faults were not found at the earliest possible phase.

Each level having its own strengths and weaknesses makes the process of selecting the appropriate level harder. To take advantage of the effectiveness of both levels, a new type of test was introduced. As suggested by the study made by Elbaum et. al. [23], differential unit tests are hybrids that retain the ability to find faults on the system level, and the advantages of automation and faster execution gained from unit testing.

One study discusses issues in empirical studies that deal with software testing techniques, where the main challenges were identified as fault seeding, the setting of the study (academic vs. industrial), replication and the involvement of human subjects [22].

3
Study Design

The study goal is to develop a framework for characterization of test cases which employs concrete and objective metrics to determine the level of a test. Using the developed framework, we can determine which testing level has the most fault detection power. The study is guided by the three Research Questions presented in chapter 1.4.
In this chapter, the data collection process will be described as well as the analysis process. The development of the framework is an extensive process so we dedicated a full separate chapter for it 4.

3.1 Data collection

In the following section the data collection process is presented in detail.There are two main sources for data collection, GitHub and Defects4J. GitHub is a hosting service for software based on Git, from where the code of the sample projects can be cloned and used to retrieve the test data dictated by the framework. "Defects4J is a database and extensible framework to enable controlled testing studies for Java programs" [26], and it can provide data about bugs and the tests that are linked to the bugs, which would be called triggering tests. For both sources, we provide information about what data is collected, from what source, in what mode and the importance of the data. Furthermore, we present the exclusion constraints for data as well as the project selection process.

3.1.1 Defects4J data collection

What? To be able to analyse the fault detection behaviours of the specific level of tests we need data about the bugs found in the open source projects and more importantly which specific tests detected the bugs. For the specific tests returned by the Defects4J tool, it would be recorded the name of the test, the name of the class that the test belongs to, the name of the package of the class, the bug id and the issue number for the specific bug.
Why? This information will be used to answer the RQ3, as the tests relevant to the existing bugs would be classified using the developed framework and then quantified to reveal which level of testing detected a higher number of bugs, hence which level has a higher fault detection power. Some of the information would be needed in identifying the tests returned from Defects4J with the tests in the current version of the projects. It should be noted that a bug can be linked to multiple

tests. For this reason we can analyse the tests that discovered the same bug and make a judgment on whether the bugs were discovered by only one level, or there were tests on different levels that uncovered the errors.
How? Defects4J was used to collect this information. Using the command-line interface that the tool, we gather the information about the bugs. Each bug registered in the tool was fixed by modifying the source code and in one commit, has an issue in the corresponding tracker and has at least a triggering test that failed in the buggy version and passed in the fixed version.

3.1.2 Test data collection

What? Because our aim is to analyse the test data from the open-source projects, we need the test suite for the projects. To be able to apply the framework we need specific information about each test in the testing suite. The basic information collected would be the name of the test, the testing class that the test belongs to, the package that contains the class of the test. For each test in the testing suite we will collect the metrics described by the framework. For the size of test characteristic, we will collect the number of methods called in the body of the test, the number of classes and packages, that are reached from the body of the test. For the size of code tested, we will collect the number of methods, classes and packages reached from the test at a depth level of two, meaning we count the metrics for the body of the test then we enter in each method that is called from the test and is part of the project and we count the number of methods, classes and packages, then sum it up to determine the final value for each metric. In this context, depth level one means the body of the test, while depth level two means the body of the methods reached fro level one. For the size of the setup, we will collect the number of methods, classes and packages reached in the setup of a test if the test class contains such a phase. This setup phase is identified by the use of the following annotation: @Before, @BeforeAll, @BeforeClass, @BeforeEach or by other configurations detected by naming conventions such as "setup", "Setup", "setUp" for the setup method. We further collect for each test if there is mock usage in the test. Based on the formulas described in the framework, we will compute a testing score which would be then mapped on the level scale. More details about the testing score and how it is calculated are provided in section 4.3.
Why? This information will be used to answer RQ2 and RQ3. This information will be fed into the framework to determine the level of the tests contained in the analysed projects. With this information we can infer which level is more prominent in open source projects. This information, coupled with the data gathered from the Defects4J tool would be sufficient for making a conclusion about the fault detection power of the concerned testing levels. Moreover, we can comment on the usage of mocks and set-up phases, as an accompaniment to the RQ2 on testing customs in open source projects. The basic information about the test would allow for an overview of how many tests a project contains and the number of test classes in a project.
How? The projects would be collected directly from their GitHub repositories. To gather the rest of the information, we will have to go through each test individually.

Information regarding the number of methods, classes and packages will be gathered using an IntelliJ plugin called SequenceDiagram [36], for all three size characteristics. The setup methods are annotated, tagged or configured, so the identification would be an easy process. For the mock usage, first the imports for the classes would be checked for mocking libraries and if present, for each test in the respective class it would be checked if mocks are used in the body of the test.

3.1.3 Data exclusion constraints

Any tests that were able to be run from the IDE were taken into consideration, regardless if they had been annotated with the @Test annotation. The tests that were commented in the code and the ones that were annotated with the @Disabled or @Ignore annotations were not collected, as well as tests that were disabled through other settings. If a test class was annotated with the @Disabled or @Ignore annotations or disabled through other settings, all tests of the class and the class itself were not counted. Test classes that contain no active tests were not counted to the total number of test classes. Similarly, packages that contain no classes that have active tests were not counted towards the total number of packages. The main methods created to run standalone tests were not counted. Another type of tests that were excluded from the analysis was benchmark and performance tests as there are considered a different category of tests.

Due to lack of time, only the current revisions of the projects were analysed. If all the revisions would have been analysed, then there would not be any reason to exclude this test, as they would be relevant for the bugs and still active. Because the analysis is only on the current revision, there is the risk that either the code or the test would evolve beyond the initial level of tests. In that case, there would be tests that do not accurately represent the level of testing intended. The projects and the tests evolve during the development period, so it is understandable that the contents of a test can change, the location of a test can change or because of other development directions a test is not needed anymore so it is removed from the suite. Taking this fact into consideration, the risk is minimized by the exclusion of tests that are not found in the current version under the same name and location or an easily identifiable location aided or not by an issue tag. If the code and consecutively the tests for said code would evolve to that of a high degree that the level of testing would be change, then it is safe to assume that the scope of the test would change along with its name and possibly its location in the testing suite. By having only tests that are present in the current version of the code, we can correlate the result of the research question 3 with the results for the research question two, to be put in perspective, considering that the tests used for RQ2 are the ones existing in the current version of the projects.
Moreover, exclusion conditions of the projects' tests are applied also to the tests collected from Defects4J, so if a test that uncovered a bug is disabled through any setting, it would not be counted. The bugs that were flagged as deprecated in the Defects4J tool were not considered, because of the fact that the deprecated bugs are bugs that can no longer be reproduced due to behavioural changes introduced

under Java 8, as stated in the README file of the Defects4J tool [50].

3.1.4 Sample

The population for the study is represented by the Java open-source projects containing tests, as this is the main focus of the study. In regards to the research questions, for RQ1 the framework for categorization of tests is developed independently of the projects chosen for the analysis. Research question number two is concerned with the level of tests and their usage in the aforementioned Java open-source projects. The data needed to respond to this question is the actual code of the projects, however the constraint of having the code available is already included in the population, covered by the open-source aspect of the projects, hence there is no need for adding any restriction to the sample for research question two. For research question number three, which aims to discover which level of tests has the stronger fault detection power, the data needed for a response is more specialized. To collect the necessary data, Defects4J tool was employed. This fact brought the implied limitation that the sample had to be chosen from the projects available in the Defects4J tool, which can be seen in table 3.1.

Table 3.1: Defects4J projects [50]

Id	**Project name**	**No. of bugs**	**Active bug ids**	**Deprecated**
Chart	jfreechart	26	1-26	None
Cli	commons-cli	39	1-5,7-40	6
Closure	closure-compiler	174	1-62,64-92,94-176	63,93
Codec	commons-codec	18	1-18	None
Collections	commons-collections	4	25-28	1-24
Compress	commons-compress	47	1-47	None
Csv	commons-csv	16	1-16	None
Gson	gson	18	1-18	None
JacksonCore	jackson-core	26	1-26	None
JacksonDatabind	jackson-databind	112	1-112	None
JacksonXml	jackson-dataformat-xml	6	1-6	None
Jsoup	jsoup	93	1-93	None
JxPath	commons-jxpath	22	1-22	None
Lang	commons-lang	64	1,3-65	2
Math	commons-math	106	1-106	None
Mockito	mockito	38	1-38	None
Time	joda-time	26	1-20,22-27	21

The Defects4J tool includes information about 17 Java open-source projects. The data needed from the tool are the number of bugs for the projects, and the additional information about which tests discovered the specified bugs. There are two projects that contained less than 10 bugs. This could have been an exclusion criterion, as less than 10 bugs would not bring enough information in proportion to the size of the testing suite, but the fact that a bug can be discovered by multiple tests opens the possibility that the two projects could bring more information than is suggested. The Closure project contains the most number of bugs, 174, which can provide extensive information for the RQ3. Despite this aspect, the project was rejected due to incompatibilities with the gathering data solution. The Closure project requires Bazelisk and integration with Bazel [51] in the IDE. Considering the fact that having more data, hence more projects to be analyzed would increase the chances of generalizability, the rest of the 16 projects were all chosen as the sample for the project.

3.2 Analysis Process

For the purpose of the study we build two automated tools to extract the needed data. The analysis process for the projects follows a 4 step plan: build, detection, acquire, analysis.
The first step is **build**, which require to successfully build the projects. The repositories of the projects would be cloned and build using the required build system. This step is necessary as the following steps would not work on a broken build.

The **detection** step goes into each successfully build project and retrieves a list with all the tests that can be run, along with information about location and provenience (name of test, name of test class, name of package, name of project). At this step we would apply the exclusion criteria for the tests that should not be included in the analysis. Additionally, at this step a list with all setup phases would be retrieved. Furthermore, a list of classes that import mocking libraries would be compiled.

With the list of tests for each project, the **acquiring** step uses the SequenceDiagram tool on each test method. We need collected the number of classes and packages reference in the tests that are native to the project, while we need the total number of methods called regardless from their provenience. This can be achieved by applying the SequenceDiagram tool with different settings, once with displaying only project classes and one with displaying all classes. We need this information for the body of the test and the code that is tested in the test, which can be achieved by the call depth settings of the SequenceDiagram tool. To acquire all the information we have to combine all the mentioned settings, so the SeqenceDiagram will be applied 4 times on the same test, resulting in 4 files containing the needed information. The process is automated. A similar but simplified process would be applied based on the setup list retrieved at the previous step.

Having the 4 files per test method, we can move to the last step, **analysis**. One of the automated tools extracts the information from all 4 files linked to a test, for all the identified tests from a project and aggregates it into a database. Separately, the tool extracts information from the files generated about the setup phase. Because the setup is for a class, the tool would register all the necessary information about the setup (methods, classes, packages) and would update all tests that are contained in that specific class with the information. For each project we would manually verify the classes that contained mocking library imports and would update the entries for the tests that were identified as using the mocking library. With all the values for the metrics available, we can apply the rules of the framework to determine the level of each test. With this current information, we can count the number of identified low level and high level test and answer to the RQ2. The data would be imported into R to perform graphs and further analysis.

To respond to RQ3 we need additional information provided by the Defects4J tool. The tool provides information about each bug and the triggered tests linked to it. We will compile a list with that information which would be subjected to the exclusion criteria. The remaining triggered tests would be linked with their entry into the database and flagged as triggered tests, adding to the entry the ID of the bug or bugs it uncovered. With this information we can quantify the number of triggered tests that are low level and high level respectively and put it in proportion with the total number of tests per each level to draw a conclusion about the fault detection power of each level.

4 Framework

In order to classify tests from low-level to high-level, a framework is needed that uses concrete and relevant metrics to place the tests on a scale so that we can categorize them. These metrics have to be measurable and have to accurately reflect the appropriate test levels.
Multiple methods for classification of tests have been explored in previous work. Orellana et. al. [41] based the classification on naming conventions and usage of different Maven plugins for running different level of tests. This approach is a subjective one, as the developers who wrote the tests decide the level of testing and then apply the naming convention and what type of plugin would run the test. Ultimately, the developer establishes the level of the test, and the further analysis of the projects is based on the bias brought by the developer. On top of this introduced bias, as one developer can classify a test on a different level than the classification made by another developer, it was found by Trautsch et. al. [12] that the term of "unit test" is used by developers for tests that do not subscribe to the definition for the term. In consequence, for the classification of tests, the framework should be based on objective metrics.

To be able to determine what metrics can be used, we have to analyze definitions for the respective test levels. Additionally to the definition for unit testing and system testing, we will include the definition for unit as it will provide a better understanding of what is tested. For this purpose, the definitions chosen for analysis are from the ISTQB glossary and IEEE 24765:2017 standard. The definitions for unit testing as presented in the two mentioned standards seem to not be in line with the modern software development context [10], so this term will be discussed in the context of object-oriented programming. From the definitions, defining characteristics would be extracted and evaluated based on importance and in relation to the test levels. Measurable metrics that describe the selected characteristics will be discussed and picked to be used in the framework.

4.1 Definitions and Characteristics

ISTQB(International Software Testing Qualifications Board)
The ISTQB uses "component" as a synonym for unit, thus in the definitions containing the respective word it refers to unit. For the purpose of the study, the definition for unit test framework is included as it would provide a better understanding of

Table 4.1: ISTQB definitions [9]

Term	Definition
component	A part of a system that can be tested in isolation.
component testing	A test level that focuses on individual hardware or software components.
unit test framework	A tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers. It also provides other support for the developer, such as debugging capabilities.
system testing	A test level that focuses on verifying that a system as a whole meets specified requirements.

what unit testing is in context.

ISO/IEC/IEEE 24765 Systems and software engineering — Vocabulary

Table 4.2: IEEE 24765:2017 standard definitions [8]

Term	Definition
unit	1. separately testable element specified in the design of a computer software component. 2. logically separable part of a computer program. 3. software component that is not subdivided into other components. 4. distinguishable architectural unit with individual identity, boundary, and behavior that is observable through interactions with other such units. 5. piece or complex of apparatus serving to perform one particular function. 6. software element that is not subdivided into other elements.
unit test	1. testing of individual routines and modules by the developer or an independent tester. 2. test of individual programs or modules in order to ensure that there are no analysis or programming errors. 3. test of individual hardware or software units or groups of related units.
system testing	testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

The IEEE definition for unit test framework ("unit test framework 1. environment that facilitates unit testing" [8]) was not used as it did not provide any new or relevant information about the context of unit testing.

When discussing the meaning of a unit, the programming paradigm has to be taken into account. A unit can have different definitions based on the programming paradigms employed. The projects to be analysed are Java projects, which is an object-oriented programming language, thus going forward the term "unit" would be discussed in an object-oriented paradigm relation. In this context, the candidates for a unit can be: packages, types (classes, structures), members of types (methods, functions, procedures), commands, expressions or blocks [14, 39]. A command, expression or block would not be considered as a unit because it is not a distinguishable architectural element with individual identity, boundary, and behavior, as per the fourth definition of unit from IEEE standard as seen in Table 4.2. A package would not be considered as a unit because it is too broad and it can be subdivided into other elements such as classes, which goes against the third IEEE definition for unit. Methods can be considered as units because they are the smallest logically distinguishable element, that cannot be subdivided into other elements. The problem with this comes from inheritance and polymorphism in object-oriented paradigm. A method in the parent class that is used in child classes should be tested in the context of each child class, as it can vary based on the attributes and operations of the child class. Thus, we chose the encapsulated class as a unit in the context of object oriented programming.

From the definitions of "unit" and "unit test" from Table 4.1 and 4.2, the elements that are defining this level of testing are: isolation, individual units, not subdivided into other elements. Because in unit testing we verify only the behaviour of the unit that is tested, any external dependencies such as inter-system communication or out-of-process dependency should be isolated and the environment should be faked, while depending on case, intra-system communication can or cannot be isolated. Regarding the criteria of the unit being tested not to be subdivided into other elements, it was established earlier that in the object oriented context the class is considered as a unit, but indeed a class is divisible into its methods and properties. Because what is tested in a unit test is an individual unit, the elements inside the test script should be part of only one class. When a class is tested, what is actually verified is its functionality. So considering this, a test that contains only calls to methods of only one class would be considered a pure unit test. Considering this, the size and reach of a test can be an indication for the level of the test, as well as the size of the piece of code that is tested.

We changed the meaning of unit to fit the object oriented paradigm and so the meaning of unit testing changed, because the definition for it relies on the term unit. The concept of system testing has the same meaning in the object-oriented context as it has in the definitions for the term, as a whole, complete, integrated system does not change its meaning regardless of the paradigm of the programming language it is written in. From the ISTQB definition of system testing, two important parts are identified: system as a whole and meeting specified requirements. These elements are consistent with the IEEE definition. First, for a test to be a system test, the system under test must be a complete and integrated system, as opposed to an isolated component in unit testing. This opposition can help in identifying the type of level a test is if we can find an appropriate metric to measure the level of isolation. In regards to the fact that a system test is verifying a specified requirement, this aspect cannot easily be checked without having access to the requirement, which can be documented through a requirement document or user stories in an Agile context. Moreover, the process for gathering requirements in open source projects is different from closed-source projects. Open-source projects benefit from a large user feedback that provides request features and report bugs or fixes needed to the code base. Based on the number of users active to a project, the main developer can choose himself what are the requirements for a project with a small number of users, without a clear direction or documentation. Another possibility for the provenance of the requirements for open-source projects might be a corporation that pays for a certain feature to be implemented. On account of this, we cannot identity any characteristic that would discern different levels of testing from the requirements. One thing that we can infer from the fact that the system testing verifies a specified requirement is the size of a test, as verifying a real user scenario with real data and resources translates to a large test. This is in opposition of unit test, where the test size should be small.

The attributes extracted from the definition and discussed above are presented in the table 4.3.

Table 4.3: Attributes based on definition

Term	**Attributes**
unit	small, not subdivided into other units
unit test	isolation, individual unit tested
system test	integrated system, complete system, test requirements, large

4.2 Metrics

From the attributes we can determine two characteristics that are defining for the two levels of testing: size and isolation. Isolation refers to isolation of the code to be tested from other dependencies. Isolation was chosen as a characteristic because it is a requirement for unit testing while system testing is done on a complete and integrated system, so isolation is not needed. This characteristic is in opposition for the two levels of test, so it can provide an initial assessment for a test position on the scale. This characteristic on its own was deemed not enough for a concrete characterization, as the presence of isolation dependencies can indicate a unit test, the absence of isolation does not provide a definite answer regarding the level, as both unit and system tests can have no isolated dependencies. For a unit test, this scenario would happen if the unit under test does not have outside dependencies, due to high cohesion and low coupling. Another defining characteristic is size. When talking about the size, we make reference to two elements: the size of the test and the size of the piece of code that is tested. Size of the piece of code tested was chosen because unit tests should test a small piece of code while system tests verify a requirement that usually is more complex and big. Size of the test was chosen as a characteristic for the same reason, a unit test should not have a big body or a complex setup, while system tests are complex and require more complicated setup.

4.2.1 Approach for size

There are different possible candidate metrics for the size characteristic. We consider this characteristic for the test and for the code that is tested.
When talking about the size of an element in software terms, the first instinctual metric is the source lines of code (SLOC). This metric is a quantifiable one and it is easy to measure. This metric has two types: the logical lines of code (LLOC) and physical lines of code (LOC). The LLOC are specific to the language of the project and it represents the number of executable statements in the code, while physical lines of code are exactly the number of text lines of the source code. The problem with the physical lines of code as a metric is the amount of noise that is introduced in the code that would be counted. Through the word noise we are referring to language keywords (such as { and } , else), commented lines, blank lines. These lines are dead lines, which do not bring relevant information for the assessment. Another problem with physical lines of code is that a piece of code can be written in a different number of lines by different developers, and with the help of lambda expressions, operation on data sources or collection can be physically reduced and compressed. Moreover, the same piece of code can have a different number of physical lines of code depending on the language it is written. Considering the limitations of the physical lines of code and the fact that the framework should work the same regardless of the programming language of the analysed project, this metric would not be part of the categorization framework. The logical lines of code suffer from the same dependence on language, but it does not count the noise that is encountered in physical lines of code. The logical lines of code are counting the declarative lines as well, which are of no concern for us and are regarded as noise. This metric is interesting but in the current form is not sufficient.

An interesting metric that is closely related to logical lines of code metric is the number of methods that are called. For the size of the test, we refer to the number of methods that are called from the body of the test. This would ignore simple declarations of variables which would be included in the LLOC metric. Because the tests are verifying the methods of classes, the number of methods called inside a test would accurately represent the size and reach of a test. We are interested in the total number of calls to the covered methods, so we include in the count all the calls to any method, regardless if it was previously executed or not. By methods called we refer to all the methods inside the test regardless if they are executed at runtime or not (ex. they are in a conditional statement). All the methods would be counted, regardless if they are part of the project or are from libraries or third party systems. This approach is closer to the one employed by Kanstrén in his paper [39] where the testing level was defined by the number of units of code exercised in the test case, with the exception that he was counting all the methods reached from the test to an end of call depth. The number of methods called represents most accurately the size of the test, but we feel it is not enough for a concrete categorization, as we lack the information about the size of the code that is tested, information that is needed to make the assessment if a test is or is not a unit test.

Because the class is considered the unit in object oriented paradigm and in a test there can be multiple methods called from different classes, we supplement the number of methods metric with another: the number of classes reached from the test. The methods counted are part of different classes, which should be counted as a separate metric. This will allow the framework to decide if a test is a pure unit test, as discussed previously, by calling only methods from the same class. Because the methods called in the tests can be from outside projects or libraries, there are actually two numbers to consider: the number of classes from the current project that contain the methods called from the test body and the number of total classes (inside or outside of the project) that contain the methods called from the test body. The condition for the pure unit test is to have methods only from one and the same class. This can be verified by checking the number of classes that the methods from the code to be tested came from. Because this is the only concern regarding the classes reached when testing, the total number of classes reached was dropped in favor of only the number of classes reached, classes that are part of the project. Hence, another metric for the framework would be the number of classes belonging to the project that are reached by the called methods from the body of the test. The number of classes reached would provide an overview into the size of the test in regards to dependencies of elements on other classes. A higher number of classes would indicate a higher-level test while a lower number of classes would indicate a low-level test. These indications are not definitive as the metrics will be combined to provide a more accurate identification.

Another metric that can complement the number of classes in regards of the reach of the test can be the number of packages that the reached classes are part of. Packages are folders or name-spaces where resources, classes and interfaces are organized as a collection that is related by functionality or scope. As with the number of classes, the interest is in the packages that belong to the project at hand, not packages from outside libraries and third party projects. This will provide an overview over the different elements of a system that are reached by the test. A higher number of packages would indicate that the test goes through multiple different modules or functionality, so for a categorization point of view, a higher number would translate to a higher-level test, while a low number of packages would indicate a low-level.

The number of methods, classes and packages were discussed in the context of the test itself. These metrics are not fully representative for the tests, as the test environment has to be prepared and configured, fields must be initialized in the test class or intensive activities are executed before running a test. Usually, these actions are performed in a set-up method that is run before the actual tests. These set-up steps should be counted as well, because they are part of the process of testing a piece of code that requires more configuration. To count these additional steps for the test, the number of methods called, classes and packages reached from the set-up method will be recorded. This set-up method can be run before each test in a test class or only once before all the tests from the test class. Regardless of the type of setup, it will be counted and recorded for each test of a respective class that contains a set-up method. In opposition to the set-up phase, a tear-down method can be run after the execution of tests to execute a clean-up of the test environment, delete temporary data, restore default values, release resources. In the case of the tear-down, the actual test was executed and all the element in the tear-down are either cleanup, or setting up defaults and temporary data for the next tests, but have no relevancy to the actual test or what it was tested. The methods called in the tear-down phase, as well as classes and packages reached could have been counted as a metric, but were not chosen for the framework because they are not related to the test or what is tested, contrary to the set-up phase, which sets up the environment of the test and without which the test would not be executed properly.

The previous discussion was held in regards to the body of the test. The same metrics can be identified for the piece of code that is tested and are based on the same reasoning. The complication with this approach is the fact that a test can verify more than just the result of a method, such as a result of a flow of methods. This makes it hard to identify exactly what is the piece of code tested from the other elements of the test. For this reason, the number of methods, classes and packages will be counted based on the body of the test on a depth call of two in order to reach the elements that are tested. This means that for the three metrics, we count the methods, classes and packages covered in the body of the test, then we enter each method called in the test, and count the methods, classes and packages reached from that method, which is consequently added to the total number for that specific metric. This would encapsulate the values chosen for the size of the test itself. For this reason, these three metrics can actually stand on their own in the framework, replacing the ones for size of test. If only these metrics would be chosen, along with the metrics for set-up size, the framework would lose the distinction between what is tested and how the test looks like. We can make a distinction between tests that are small and are testing a small piece of code, tests that are small but verify a large piece of code, tests that are big in body but check a small piece of code and tests that are big and verify a big piece of code, in regards to the characterization of a test on the high-low level scale. In the interest of not losing a piece of information that can help more accurately describe the place of a test on the scale, both the metrics chosen for the size of the test and the ones chosen for the size of the piece of code tested would be taken into consideration in the framework. For the case of the piece of code tested, as no element that is tested is outside the test there is no need to take into consideration the set-up or tear-down phases.

Another metric that was considered was the complexity of the test code and the code that is tested, as it would count the interactions between entities. The complexity metric is related to the size, as a size increase or decrease requires respectively an increase or decrease in complexity. Ultimately, this metric was not chosen as the relationship between size and complexity is not a strong one, and the complexity would not bring information strongly related to the level of a test.

In this section we discussed some metric candidates for the size characteristic, which are summarized in Table 4.4. In the end, for the size characteristic we chose the total number of methods, number of intra-project classes and number of intra-project packages reached from two sources: the body of the test and the code that is tested. Our approach for the number of methods metric differs from Kanstrén by the level of call depth that we measure, as we only go to a call depth of two while his approach counts the methods until they reach the end of the call depth.

Table 4.4: Discussed metrics for size

Metric	**Status**	**Reason**
SLOC	Dropped	Noise
LLOC	Dropped	Noise
No. of Methods	Selected	Represent size of test and code tested
Total no. of Classes	Dropped	Unnecessary count of outside project classes
No. of Classes from Project	Selected	Represent reach of test/code tested/setup
No. of Packages from Project	Selected	Represent reach of test/code tested/setup
Cyclomatic complexity	Dropped	Not strongly related to size

4.2.2 Approach for isolation

To measure the isolation level of a test, one metric can be usage of test doubles in the analysed test. Five types of test doubles were introduced by Meszaros in his book [42] and adopted by the testing community. These categories are stubs, mocks, spies, fakes and dummy. Although in an ideal context, the number of dependencies for the piece of code tested would be counted and then compared with the number of dependencies that were isolated by test doubles, this would be a complex process that would take more time than we can assign for. Moreover, the identification of fakes and dummy test doubles is complex. One can use the naming conventions that adds to the element that is wanted to be doubled the prefix Fake or Dummy to identify these test doubles. The problem with this approach is that there is no knowledge on whether the developers are applying correctly and consistently on all accounts the respective naming conventions. Because the framework should be based on objective metrics, the usage of fake and dummy type of test doubles would not be taken into account. For the other three types of test doubles, they can be identified by the usage of mocking frameworks. Although there are named mocking frameworks, those frameworks have implementations not only for mocking, but for spying and stubbing as well. As mentioned before, we are unable to count the dependencies that remained un-isolated and the dependencies that were isolated through mocking, stubbing and spying. For this reason, the metric of mock usage cannot be a quantifiable one, instead it will be a logical one. A test either contains mock usage or not. For the purpose of classification, we acknowledge that a quantifiable metric would have brought more information that the classification could use to make a decision, nonetheless the logical metric is not the only metric that the classification is based on so we absorb the risk of using it by using the metrics for the size characteristic as being more important of a deciding factor.

In the table 4.5, we aggregated the metrics discussed in this chapter for their respective characteristics.

Table 4.5: Characteristics and metrics

Characteristic		Metric
Size	Size of test	Number of methods
		Number of classes
		Number of packages
	Size of code tested	Number of methods
		Number of classes
		Number of packages
	Size of setup	Number of methods
		Number of classes
		Number of packages
Isolation		Mock usage

4.3 Test score and Scale

The respective metrics would be measured for each test and the same formula of categorization would be applied in order to map the scores to the respective test levels. A pure unit test would be a test of one method, from one class, where the method uses only methods in that respective class, regardless of mock usage. In this context, an integration test would be a test that contains multiple methods from two or more different classes, though we are not focusing on this level. This definition of a unit test is too rigid for the current context of the industry, so we broaden the scale for the unit test. The percent of pure unit tests can be presented in the results chapter for RQ2 as an addendum. This will mean that the level of integration would be enveloped in part by the unit level and partly by the system level. In this way, the scale would accommodate only the two levels of testing relevant to this research. Although the seven metrics identified for the framework are needed for the final categorization, they do not provide the same value or power in the process of classifying the tests. Between the size of test, the size of code tested and the size of the setup, the most valuable metric is the number of methods, followed by number of classes then number of packages, and it should be reflected in how the score is calculated for each test.

To be able to make conditions on the score of a test we must analyse the power of each metric and the different combinations that can occur. Regarding the size of the test, a higher number of methods from the same class indicates more to a low-level test than a smaller number of methods from different classes. Likewise, a higher number of classes from the same package indicate more to a lower-level than a lower number of classes from different packages. For the size of the code tested we must keep in mind that the metrics enclose the size of the test as well. Due to this fact, the value limits for the score grid of these metrics should be doubled or adjusted properly from the ones used in the size of the test. To assign points for the values of the metrics we looked at the possible range for the values and how would that reflect to the level of test. For any of the size characteristic, the fact that the number of packages cannot be higher than the number of classes and the number of classes cannot be higher than the number of methods was taken in account when assigning the score to the respective values.

We explain only the reason behind the assignment of scores for the respective values of size of test, as the values for the size of the code tested are double the values for the size of test, but have the same score because this metric encapsulates the size of the test.We looked into multiple best practices for size of a method and with the assumption that on each line we would have a method call, we settled on 10 methods as a base to increment our score grid. For the baseline on number of classes and number of packages we used the definition for pure unit test, which dictates that the methods belong to only one class, that subsequently is part of only one package. We attributed a score of 1 for each baseline, so a test with 0 to 10 methods from one class belonging to one package would have a score of 1 from number of methods, a score of 1 for number of classes and a score of 1 for number of packages. For the number of methods, we incremented the values by the base value following an incrementation of 1 in the score. We could have continued the incrementation until the value 100 would be reached, but we consider that a test that has more than 50 call methods is big enough, hence we attributed a score of 10 for any value higher than 50. The score of 10 is our maximum value in the framework, and it is attributed to values that we consider big enough.

The grid for number of classes and packages follows a different approach. For this two metrics, we consider a value higher than 10 to be big enough, so we attributed the maximum score of 10 for it. We could have incremented the value by the baseline value for a score increment of 1, but then the score would have been too granular. As we do not use the score to compare tests to each other, but to place then on low or high level, we do not need that level of granularity. So between the base value of 1 and the maximum value of over 10, the inside segments of values were decided at 2, 3-5 and 5-10. This is not too granular but allows for variation. It is more grave from the test level perspective to have more packages reached than classes, so that is why for the value of 3-5 packages the score is higher than the same value for classes. The other segments of values are either considered small enough, so the reach of 2 classes can be equivalent with the reach of 2 packages in regards to the score for the specific metric, or considered big and are assigned a value of 5. The chosen scores can be seen in Tables 4.6 and 4.7.

Between the three categories for the size characteristic, the one with the lower power for test placement decision is the size of the setup. For the size of setup, we decided to not attribute a score for all metrics (number of methods, classes and packages), but we formed a condition based on their values. A setup is considered small if it contains less than 5 methods from a maximum of 2 classes, classes contained in the same package. The values used to determine the size of the setup were chosen from different considerations. The steps taken in the setup phase are initialization of data, configuring the environment or time intensive activities. Considering the best practices regarding method size and the usual actions taken in the setup phase, the limit of 5 methods for a setup to be considered small is reasonable. Regarding the class limit, keeping in mind that the pure unit tests verify the functionality of one class and the fact that usually a class is not fully isolated (has connections to

Table 4.6: Test scores for size of test

Characteristic	**Metric**	**Value**	**Score**
Size of test	No. of methods	0-10	1
		11-20	2
		21-30	3
		31-40	4
		41-50	5
		>50	10
	No. of classes	0-1	1
		2	2
		3-5	3
		5-10	5
		>10	10
	No. of packages	0-1	1
		2	2
		3-5	4
		5-10	5
		>10	10

Table 4.7: Test scores for size of code tested

Characteristic	**Metric**	**Value**	**Score**
Size of code tested	No. of methods	0-20	1
		21-40	2
		41-60	3
		61-80	4
		81-100	5
		>100	10
	No. of classes	0-2	1
		3-4	2
		5-10	3
		10-20	5
		>20	10
	No. of packages	0-2	1
		3-4	2
		5-10	4
		10-20	5
		>20	10

other classes), we consider the limit of 2 classes for a small setup is fair. As for the package limit, because the low level tests should be limited in scope, the setup phase for them should not be broad. No setup or a small setup as described above is correlated with a low level test. A setup that is larger than what was previously described would be considered for a high level test. This in itself is not a fully deciding factor, as the size of the setup on its own cannot accurately reflect the level of the test, but adding it to the score reflected by the other characteristics will improve the accuracy of the categorization. In a more concrete way, no setup or a small setup would not increase the test score, while a large setup would be worth 1 point for the total score of the test. Based on the size of the setup, the scores can be seen in Table 4.8. Similarly, the presence of mock usage would decrease a score of a test towards the lower level, while the absence of mocking would not change the value of a test score. Concretely, the presence of mock usage would decrease the score of a test by one point. The scores reflected by mock usage can be seen in Table 4.9.

Table 4.8: Test scores for size of setup

Characteristic	**Condition**	**Value**
Size of setup	No. of methods <= 5 & No. of classes < 3 & No. of packages < 2	0
	else	1

Table 4.9: Test scores for isolation

Characteristic	**Metric**	**Value**	**Meaning**	**Score**
Isolation	Mock usage	0	No mock usage	0
		1	Mock libraries used in test	-1

Based on the values recorded from the metrics, a score will be computed for each one, as shown in the tables above. To determine the total score of a test, the scores from the individual metrics would be summed.

The scoring grid allows for a limited range of values. For high enough values of the framework metric, the score was limited at 10 points, a high value in the scale. This limit was placed because the concern is with the difference between low level and high level, not how different two high level tests can be. With the current scoring, the highest value possible for a test is 61 points and can be achieved by a test that contains more than 50 methods, from more than 10 classes that are in more than 10 packages, where the code tested contains more than 100 methods from more than 20 classes contained in more than 20 packages, test that does not use mocking from mocking libraries and has a big setup phase. The lowest value possible for a test is 5, where the test is empty without a big setup phase or contains only 1 to 10 method from one class, and those methods does not contain in total more than 20 minus the total methods contained in the test, with the use of mocking inside. There are tests that contain only methods outside of the project that verify settings or other input data. For this kind of tests, because the methods are outside of the project, the number of classes and packages would be 0. Also, empty tests exist, where the number of methods, classes and packages reached is 0. The score gird could allow for the value 0 for any of the size characteristics to be linked to a score of 0, but ultimately it was chosen to link the 0 value into the first level of the grid for each size characteristic and assign it a score of 1. This decision was taken to have a consistent test scale, as if the 0 value would be associated with a 0 score, there would have been strange values between 0 and 5 and it would have been hard to associate the respective tests on low level when a true unit test would have the score of 6 or 5 is mocking is involved.

Having the scale to place the test scores in helps with determining the distribution of the tests, but additionally to the scale, there is the need of a value that can be considered the limit between the low level tests and high level tests. It is important to note that the value in the middle of the scale (30-31) does not represent the limit between lower level and higher level. The scale was limited to the highest value of 61 for convenience. Moreover, the lower limit is determined by an empty test, but due to the definitions of unit tests regarded in section 4.1, an empty test would not be considered a unit test as in fact it does not test anything. The same argument can be made for tests that contain methods from other libraries beside the project, meaning that there are no classes or packages reached from that test. This will render the score for number of packages and number of classes for both size of test and size of code tested as 4, so the total score for the test would vary with the number of methods in the test and in the code tested, as setup and mocking together can bring between -1 and 1 point to the score. These tests can populate a range of scores between 6 and 21. The empty tests are regarded as low level tests because of the 6 score, but the tests that contain only methods from outside the project will not be placed automatically on the low level, but judge by their score. Although not having methods from the project to test makes the test lighter, a test with a very high number of methods cannot be considered as a low level test. Returning to the definition of a pure unit test, a test that contains only on method from one class, and that method does not reference other methods outside of that class would score between 6 and 15 points without considering the setup and the use of mock libraries. The lower value is achieved when the method does not call to any other methods, while the higher value is achieved when the method calls 100 or more other methods from the class. The value that determines the limit between low level and high level is 15. This value was chosen as it encapsulates the definition used for pure unit tests but allows a range of combinations of value metrics that does not limit the low level to only pure unit tests.

5 Results

In this chapter the analysis results are presented and the research questions are answered. Additionally, a sub-chapter was added to present information about the testing state of the analysed projects. The tables relaying the information can be seen in the Appendix.

5.1 RQ1

In this section, the categorization framework developed in section 4 is presented as a response to the first research question *RQ1: What metrics can act as proxy for the testing levels?*. Seven metrics were identified and combined as a substitute for the testing levels. The possible values of the respective metrics were mapped to score points that were summed to provide a test with a total score that would be mapped on a low-high scale to determine the level of a test. The seven metrics are presented in Table 5.1. Details about the computation of the score based on the respective metrics can be found in Tables 4.6, 4.7, 4.8 and 4.9.

Table 5.1: Characteristics and metrics

Characteristic		**Metric**
Size	Size of test	Number of methods
		Number of classes
		Number of packages
	Size of code tested	Number of methods
		Number of classes
		Number of packages
	Size of setup	Number of methods
		Number of classes
		Number of packages
Isolation		Mock usage

The framework was applied on the collected tests from the selected projects. The framework has to be verified in order to justify the results for the remaining 2 research questions. The verification has two objectives: first objective is to validate the fact that the metrics chosen can act as proxy for the level of testing and the second objective is to detect types of test development (outside of the general way of testing) that are not compatible with the framework. A big problem in performing the verification of the framework is the source of truth. In this case, the source of truth would be the level associated with a test. To determine the level of a test, the researcher needs to apply definitions or characteristics on the test, which leads back to the problems described in the thesis: multiple different definitions for testing levels, developers using the terms inappropriately. To be able to perform the verification and handle the pitfalls mentioned, the process of categorization would be described in detail so that there would be no doubt if the categorization is valid or not. For the source of truth, multiple conditions would be taken into consideration. The definitions described in the framework chapter 4 for unit and system testing in the context of object-oriented programming would be the ones used for characterization. Another additional problem that comes from using the definition for unit and system tests is the fact that the integration level is not considered at all, so if a test is on the integration level, the characterization would not work. The framework measures the size of the test and the size of the code tested through the number of methods, classes and packages, while a developer does not need to quantify these metrics to gauge the size of the test and code tested, moreover a developer can understand what is the test and what is the code tested due to the language, commands and other additional information (like test name, comments) that cannot be introduced in the framework. Due to additional information that a developer could use to determine the level of a test, we will take into consideration the subjective characterization of the tests by the researchers. We will also consider the intended level of test gathered by checking test or test class comments or tags.

From the three characterization conditions: researchers' classification, developers' classifications and the framework classification, the source of truth would be the two out of three agreed upon level. We use the two out of three methods to remove bias. There are exceptions to this rule. If the developer characterization is missing then the characterization of the researcher would be the source of truth. Beside the concrete categorization of the framework (low or high level), we will take into consideration the score of the test as well.

To verify the framework we need to take a sample of the tests. For this purpose we employ a stratified sampling method. The population is represented by all the tests collected from the 16 projects and the first stratum is the level of tests based on the categorization framework. For this purpose, we chose a sample of 10 tests with a disproportionate sample, 5 tests from low level and 5 tests from high level. It was decided against the proportionate sampling because the number of high level tests analysed in a proportionate sample of 10 would be a very low number and would not provide us with the information needed for verification. The tests are selected at random from the sample. This would represent an initial verification. Due to the fact that different projects can have different methods of testing, another verification would be performed. For this, we will stratify two characteristics: level and project. Combining these characteristics we will have 32 groups, so we will randomly select low and high level tests from each of the 16 projects to be verified. The projects have different number of tests, so the sample should somehow be proportionate. If we would use a fully proportionate sample and take a sample of two for the project with the least amount of tests, the total number of tests in the sample would be 206, which is a big number for a manual verification. Instead, we used a somehow proportionate sample based on the number of test per project as follow: for projects with number of tests under 1000, we sample 2 test, for project between 1000-1999 we sample 4 tests, for projects between 2000-2999 we sample 6 tests, from projects between 3000-3999 we sample 8 tests and for projects over 4000 we sample 10 tests. With this configuration, the total sample would amount to 70 tests. The number of tests sampled from each project is an even number, because the sample would be taken half from low level tests and half for high level tests from the selected project. The framework evaluation results can be seen in Tab. A.2, A.3, A.4, A.5, A.6 in the Appendix A.

The first evaluation had a rate of 90%, from the 10 randomly selected tests, 9 were characterized on the correct level. The second evaluation had a rate of 71.4%, from the 70 randomly selected tests, 50 were placed on the correct level. In both checks, the rate of correct categorization for low level tests was 100 %. The inconsistencies between the framework categorization and the source of truth were attributed to the high level tests. The 20 high level tests that were categorized by the developers and/or by the researcher as low level have uncovered situations where the framework falls to properly asses the tests. Analyzing the 20 tests that were categorized incorrectly, three cases were discovered to be the reason for misclassification. First case of misclassification came from the fact that in the test, for the same case multiple values were verified. This would inflate the score based on the size of methods, so although the case tested belongs to a low level, the repetition of said case elevates the level. In this case, if the test would be broken down in multiple smaller tests for each value, they would be categorized on a lower level. This is more of a testing anti-pattern for testing then a problem with the framework. In the same category exists tests that actually contain multiple test cases inside; this is an anti-pattern and is attributed to mistakes of developers or testers. Another case or wrong classification was the private testing methods and assertions developed in the projects. In this case, the framework cannot distinguish between the code tested and the private testing methods and assertions, as they indeed are code in the project, so the framework considers them code that should be tested because they are placed in the testing methods, so they raise size characteristic by the number of methods, classes and possibly packages. As developers, it is understood that a privately developed assertion method is used in the same manner as an assertion method from the testing framework, so conceptually we understand it as a verification, without the need to look inside it. Under the same case falls the usage of private testing methods. For reasons related to reusability of code, some tests have in the body only one method, a private testing method from the same testing class. We understand that what is tested is the code in the private method, but the framework cannot distinguish that. For the framework, the private method is the code that is tested while in reality, the code that is tested is the code inside the private method. From the perspective of the framework, the usage of private testing methods decreases the size of the tests and possibly the size of the code tested. Another case for misclassification came from the fact that multiple projects from the sample are self-identified libraries and without a clear requirement document, some tests can be classified as both low or high level, because a test can verify a requirement by only using one method equivalent with a unit, but there is no integrated system to check because the project is a library. Two out of the three cases of misclassification are due to unusual styles of testing. This analysys provided us with exceptions on the type of tests that the framework would not function accordingly.

We found seven metrics -mock usage and number of methods, classes and packages for size of test, code tested and setup, that can successfully act as proxy for testing levels, as confirmed by the success rate of the two verifications performed.

5.2 RQ2

In this section, by applying the categorization framework on the collected tests from the sample projects, we can provide a response to the second research question *RQ2: What level of testing is used more in open source projects between unit testing and system testing?*. From the total of 25477 tests, 19984 were categorized as low level tests, representing 78.4% of the total, while 5493 tests were categorized as high level, representing 21.6% of the total of tests.

To conclude, based on our analysis of the 16 projects, the level of testing that is more used in Java open source projects is low level (equivalent to unit testing).

In addition to the response, we provide information about the percentage of low-high level tests per project and a comparison between the number of low level tests categorized by the framework and the number of unit tests as dictated by the pure unit test definition. Furthermore, the total distribution of scores on the score scale and the score distribution per project are presented.

The percentage of low and high level tests per projects can be seen in Fig. 5.1. None of the projects had a percent of 100% of low or high level tests. In all the projects the percentage of low level tests was higher than the percentage of high level tests. The project with the least amount of difference between percentages was Jsoup, where from a total of 862 tests, 442 were categorized as low level and 420 were categorized as high level, representing 51.28% and 48.72% respectively. The highest percent of low level tests was achieved by the JacksonXml project, where from a total of 305 tests, 292 were categorized as low level and 13 as high level, representing 95.74% and 4.53% respectively. From the 16 projects, seven had a percentage of low level tests that was higher than 90%, 2 had the percentage of low level tests between 80% and 90%, 4 had the percentage between 70% and 80%, 2 had the percentage between 60% and 70% and only one project had a percentage of low level tests between 50% and 60%.

Although some projects have a similar percent of low level tests, they vary in the number of total tests per project. The Fig. 5.2 shows the number of low and high level tests per project. It can be seen that the projects with a smaller number of tests (<2000), which represent 10 out of 16, tend to have a higher percentage of low level tests (>85%), with the exception of two projects. From the six projects that contain more than 2000 tests, only one has a percentage of low level tests higher than 90, while the other 5 projects having less than 78%.

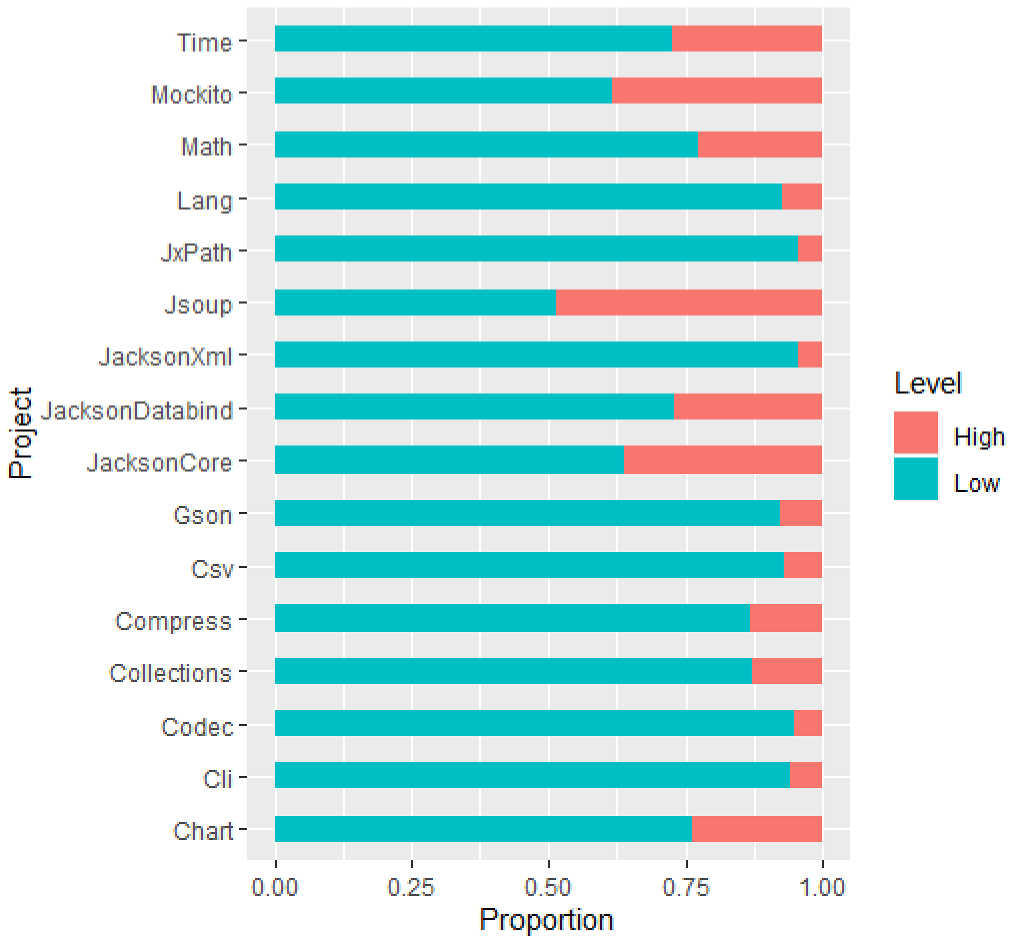

Figure 5.1: Proportion of low and high level tests per project

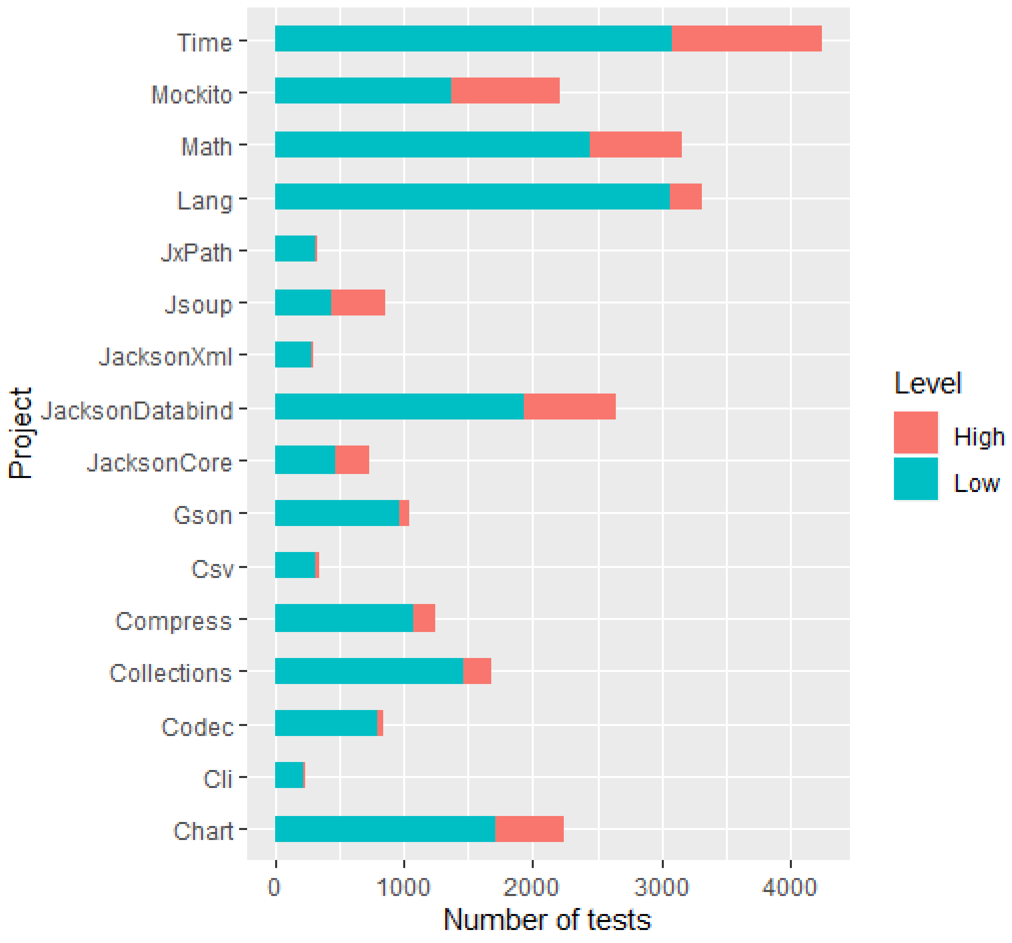

Figure 5.2: Number of low and high level tests per project

In the previous chapter, the definition of unit test in the context of object oriented paradigm was discussed. Reiterating, a test that contains methods from one class that only uses methods from that specific class is regarded as a unit test. Under this definition, from the total of 25477 tests, only 3772 tests would be regarded as unit tests, representing 14.8%. In this context, the integration level is fully enclosed in the high level, unlike in the framework, where the integration level is shared between low and high level. It can be seen in Fig. 5.3 that this definition is very restricted, as compared with the developed framework which found 19984 low level tests, in comparison to only 3772. In addition to the pure unit tests, 108 empty tests were counted, representing 0.42% of the total 25477 tests.

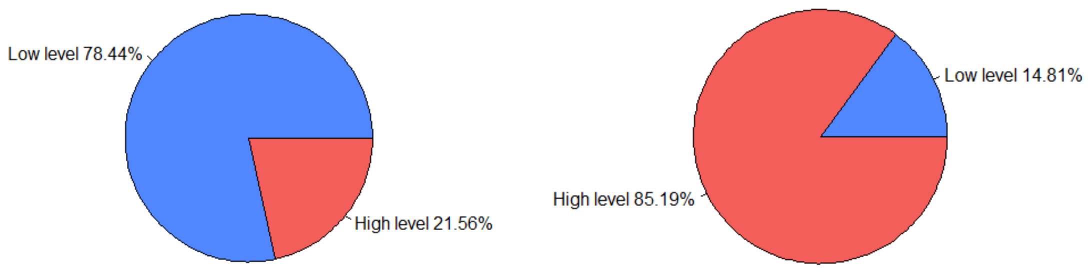

Figure 5.3: a) Low and high level test percentage vs b) Pure unit test percentage

The level of the tests are decided by the score calculated using the categorization framework. It is important to show the diversification in the score for the total amount of tests. The Fig. 5.5 shows that the score that occur most often throughout the tests is 6, with a number of 3997 tests having that score, followed by a score of 8 for 3147 tests. The Fig. 5.4 shows the distribution of scores on the test scale. Although the scale for the test score has the higher limit set at 61, the maximum score of the tests was 44, with 2 tests having this value. Only 4 tests have a score that is higher than 40, while 314 tests have score between 30 and 39. The distribution graph and the histogram show how the test scores are spread on the scale. The categorization of the tests is based on the set limit of 15 points, and with the help of the two graphs the difference in categorization can be grasped if the boundary between low and high level would have been another value.
The distribution of scores per project can be seen in Fig. 5.6. None of the distributions are symmetrical. Almost all the projects score distributions are unimodal, with the exception of Mockito, which is a bimodal distribution, with peaks at the value 6 and 16. With the exception of JxPath, Mockito and Jsoup projects, all other projects have a positively skewed or J shaped unimodal distribution of scores. Some projects like Codec, Csv, JacksonXml and Lang have strong positively skewed unimodal distributions, this mean that even if we would restrict the definition of low level test lowering the threshold of 15 points, they would still have a higher proportion of low level tests. The JxPath project has the flatter distribution, with a peak at the value 6 and the maximum achieved score of 28. The Chart, Codec, Compress, Gson, JacksonXml, Lang and Math projects have the peak at the value 6.

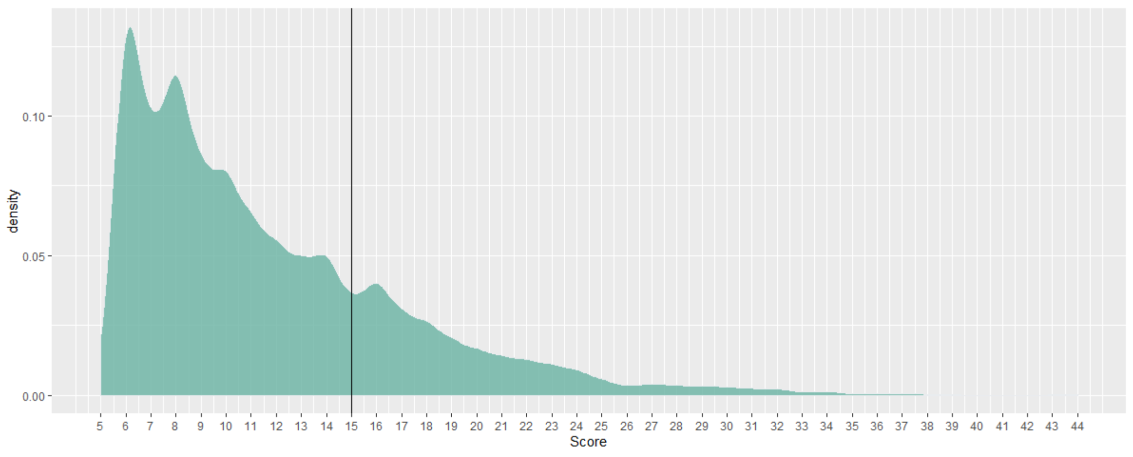

Figure 5.4: Density of test scores

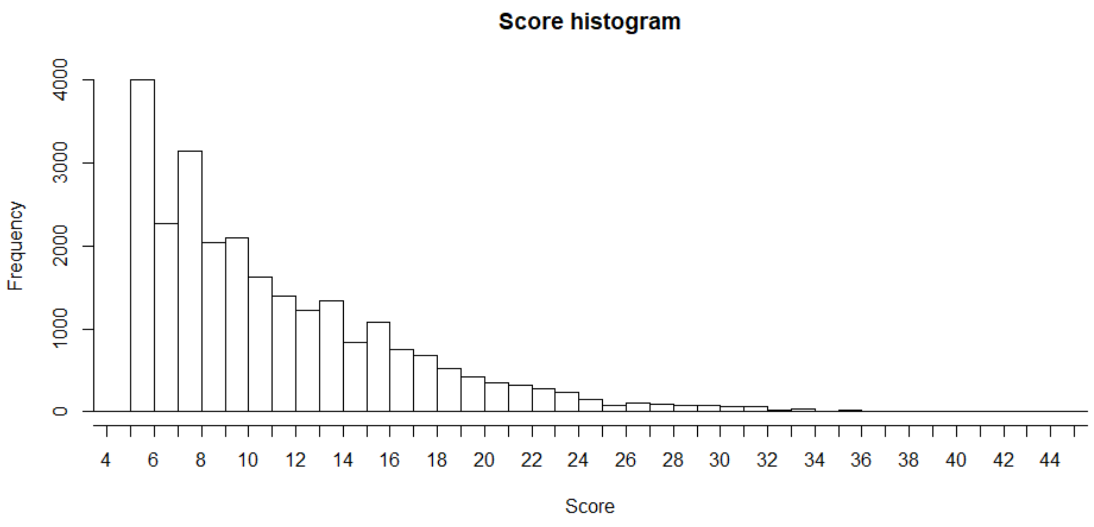

Figure 5.5: Histogram of test scores

The Cli and Csv have peaks at value 7, while Collections and JacksonCore projects have the peak at value 8. JacksonDatabind project has the peak at value 10, Jsoup project has the peak at value 14 and the Time project has its peak at value 11. For every project except Lang, the minimum score is 6. The Lang project has 3 tests that have a value of 5, meaning that to the base value for size of code and size of test, they did not have big setups and employed mocking in their implementation. The maximum score value in total was 44, less than the maximum value of the scale, which is 61. The score of 44 was achieved by tests in Chart and Mockito projects, followed by a maximum score of 43 in the Math project. The smallest maximum score was achieved by the Codec and Gson projects, with the value of 26.

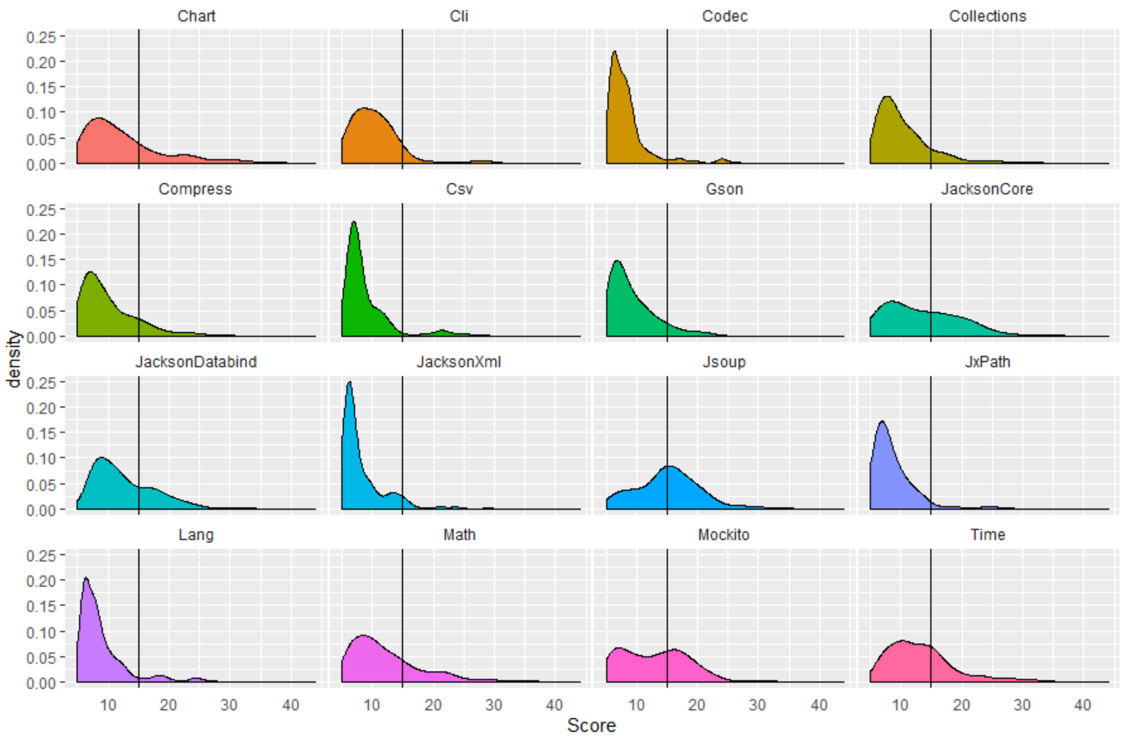

Figure 5.6: Score distribution per project

5.3 RQ3

In this section, a response would be provided for the third research question *RQ3: To what extent system testing can cover the fault detection power of unit testing ?*. First, we will provide quantitative information about the bugs and the tests that uncovered them, then based on the percentage of low and high level tests that uncovered bugs we can make a conclusion on the first hypothesis. Furthermore, we present the distribution of scores for the test that uncovered bugs from all projects and for individual ones. In addition, an analysis on the level of tests associated with bugs that have multiple tests that uncover them was performed in order to provide a verdict for the second hypothesis. We further analyse the relationship between the metrics and uncovering bugs, as well as the relationship between the metrics, uncovering bugs and test level.

The Defects4J tool contained information for a total of 661 active bugs gathered from all the analysed projects. This number of bugs was uncovered by a total of 998 tests, which represent approx. 3.92% of the total number of tests in the projects. The tests that uncovered bugs would be referred as triggered tests. The information per project about the number of tests, bugs and the number of triggered tests can be seen in Table A.1. The table contains two columns related to the number of bugs: Defects4J Bugs and Bugs. Defects4J Bugs represents the number of active bugs that the Defects4J tool have analysed, while Bugs contain the number of bugs that have at least one triggering test in the current revision of the projects, hence the number of bugs reported by Defects4J will be higher or equal than the other Bugs number. To respond to the first part of the third research question we only focus on the triggered tests and their level, but for the second part we take into account the number of bugs that have at least a triggering test in the current revision. Three out of the 16 analysed projects had a smaller number of triggered tests than the number of bugs. This phenomenon can be attributed to either the exclusion criteria for the tests (a test that uncovered a bug was later flagged as deprecated, ignored or was disabled through other methods) or due to the fact that a test can uncover multiple bugs. It should be noted that for the same project, if a test did uncover multiple bugs, that test was counted only once. The rest of the projects contained more triggered tests than the number of bugs in the project, due to the fact that a bug can be linked to multiple tests.

Table 5.2: Projects bug information

Project	**Tests**	**Triggered tests**	**% triggered tests**	**Defects4J Bugs**	**Bugs**
Chart	2244	85	3.78%	26	23
Cli	235	37	15.74%	39	28
Codec	842	37	4.4%	18	17
Collections	1680	3	0.18%	4	3
Compress	1250	69	5.52%	47	47
Csv	343	22	6.41%	16	15
Gson	1053	33	3.13%	18	17
JacksonCore	729	48	6.58%	26	23
JacksonDatabind	2652	116	4.37%	112	99
JacksonXml	305	12	3.93%	6	6
Jsoup	862	131	15.2%	93	89
JxPath	331	29	8.76%	22	22
Lang	3320	92	2.77%	64	44
Math	3168	101	3.19%	106	65
Mockito	2215	109	4.9%	38	37
Time	4248	73	1.71%	26	26

By applying the categorization framework on the respective tests collected from the Defects4J tool, we can answer the question of which level of testing uncovered more bugs. From the total of 998 tests, 656 were categorized as low level tests, representing 65.73% of total tests, while 342 tests were categorized as high level tests, representing 34.27% of the total tests. The proportion between low and high level tests per project can be seen in Fig. 5.7. Three projects contain a higher percent of high level tests discovering bugs than the low level, while the rest of the projects contain a higher percent of low level bugs than high level bugs. In three projects, the percent of low level tests that discovered bugs is 100%, followed by three other projects with percentages over 90 for low level tests. The lowest percentage of low level bugs was 39.58% recorded in the JacksonCore project, followed by two projects with percentages between 45 and 50 for low level bugs. The other 7 projects have percentages between 50 and 90. The Fig. 5.8 shows the concrete number of low and high level tests from the tests that uncovered bugs per project.

The result regarding the percent of low level versus high level tests that uncovered bugs can be attributed to the fact that there is a higher number of low level tests in the total sample of test. An analysis on the rate of uncovering bugs from the total number of bugs must be performed. The total number of tests was 25477, with 19984 low level tests and 5493 high level tests. From the 19984 low level tests, 656 uncovered bugs, which represent a percent of 3.28%. From the 5493 high level tests, 342 uncovered bugs, which amount to 6.23%. The uncovering bug rate for high level tests is higher than the rate for low level tests, to be exact it is with 89.67% higher. It can be said with a confidence level of 95% that the high level tests perform better than low level tests in regards to uncovering bugs.

The results suggest that high level testing leads to a greater fault detection than using low level testing, even though the percentage of low level triggering tests was higher than the percentage of high level triggering tests.

In the Fig. 5.9 the density of test scores for tests that uncovered bugs is shown. The histogram shows that the most frequent score for the tests that uncovered bugs is 6, with 118 tests having that score, followed by the value 8, which 110 tests had. The minimum score value was 6, while the maximum score value was 38, with only one test having that value.
The distribution of tests scores per project for tests that uncovered bugs can be seen in Fig. 5.11. The distribution of scores for Chart, JacksonCore, Lang, Math are flatter. The minimum score for all the projects except Collections and JacksonDatabind was 6. The Collection project is an exception, as it contains only 3 tests that uncovered bugs, so the distribution was not accurate due to small number of values. The three tests had a score of 9, 13 and 14, all scores being under the low level category. The JacksonDatabind project has a minimum score of 7 and a maximum score of 33.

The Defects4J tool registered a total of 661 bugs from all fifteen projects. From the total bugs, only 561 bugs had triggered tests. The difference comes from the fact that the triggered tests linked to the bugs were not present in the current revision (they were either moved or transformed to a degree that would not reflect to the initial bug) or they have been disabled or ignored through various settings. From the 561 bugs, 153 bugs had multiple triggered tests. The rest of 408 bugs had only

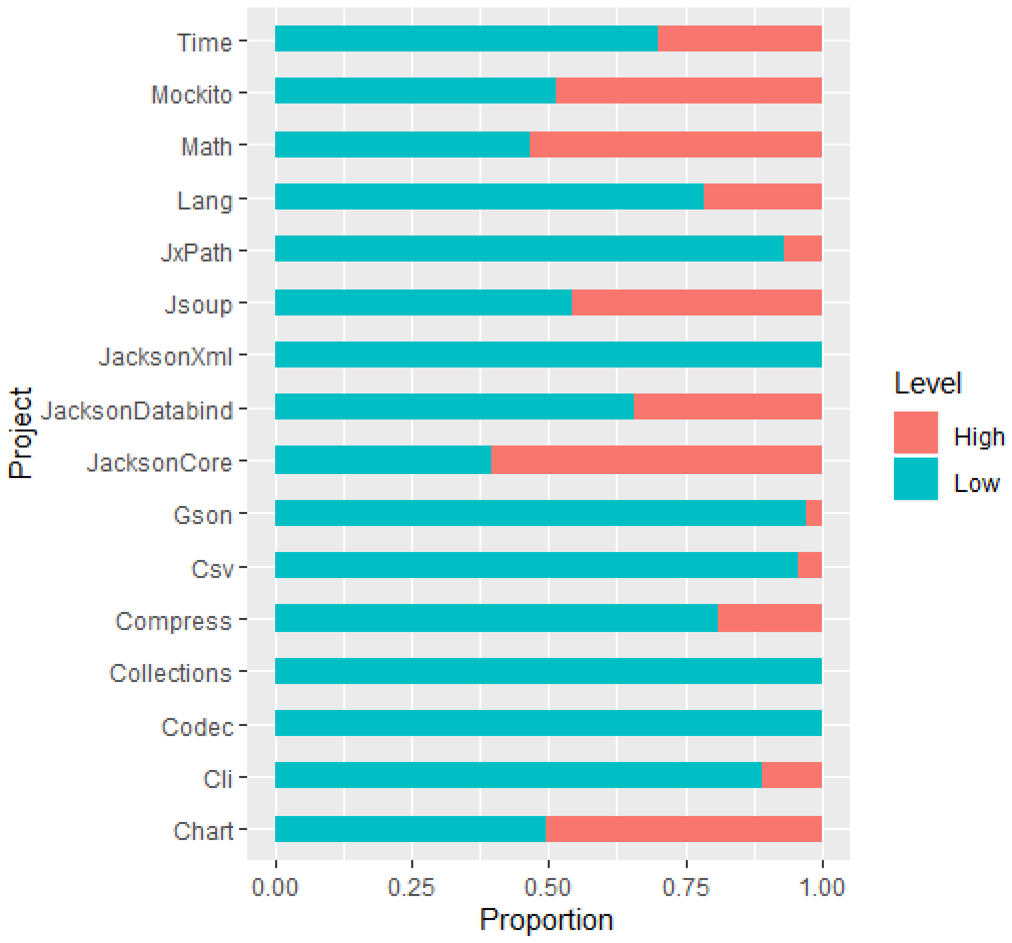

Figure 5.7: Proportion of low and high level trigger tests

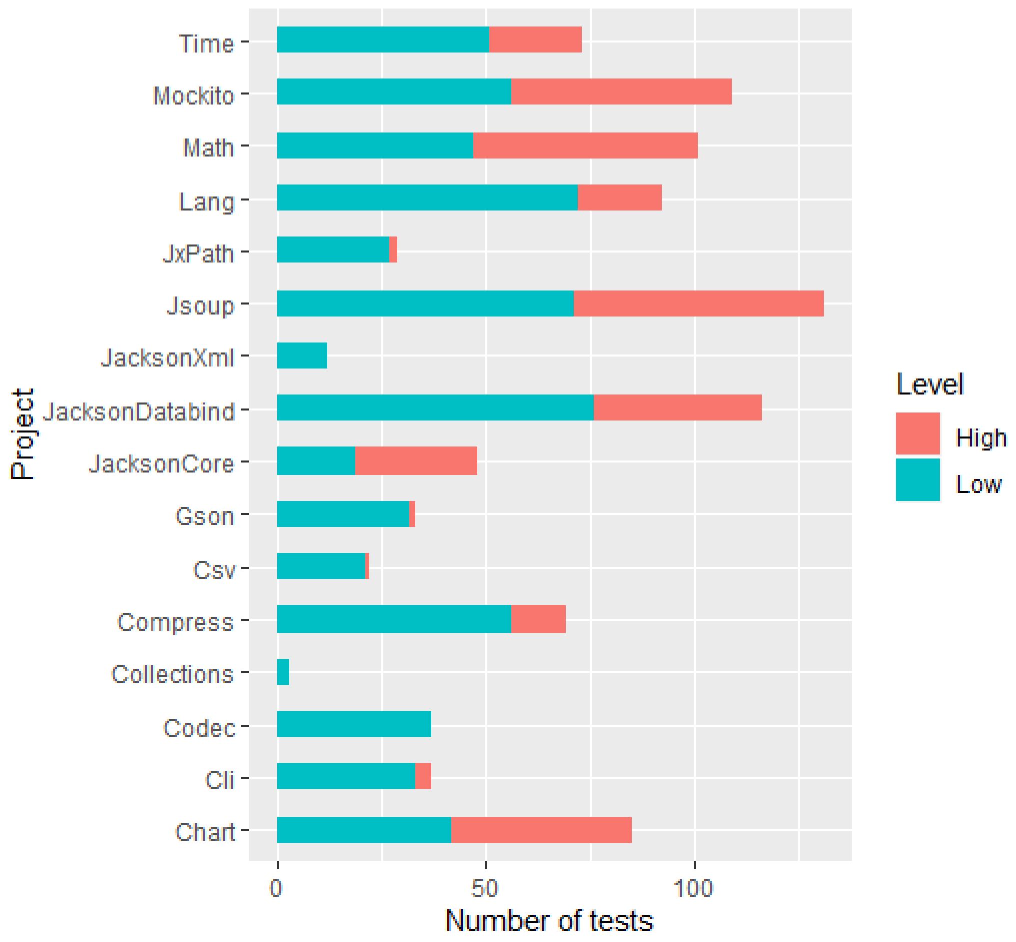

Figure 5.8: Number of low and high level trigger tests

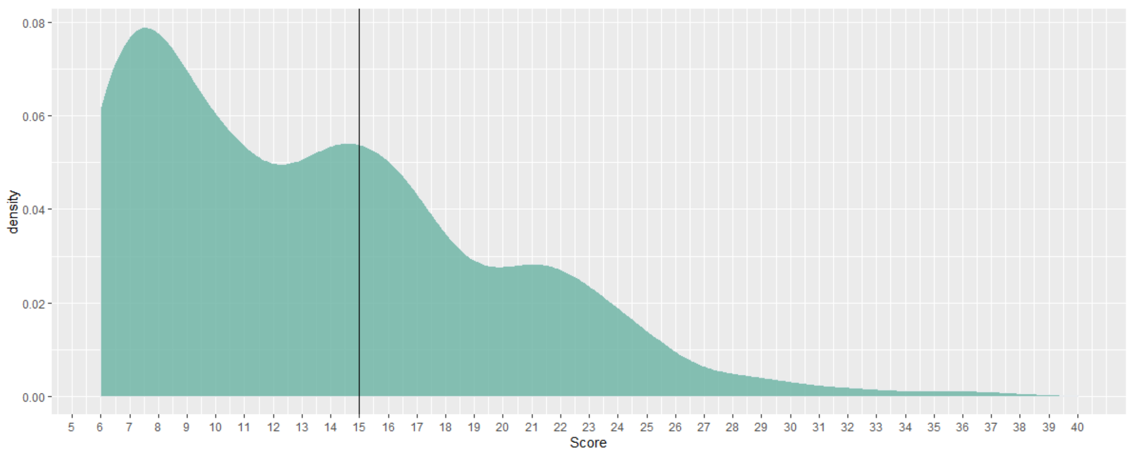

Figure 5.9: Density of test scores

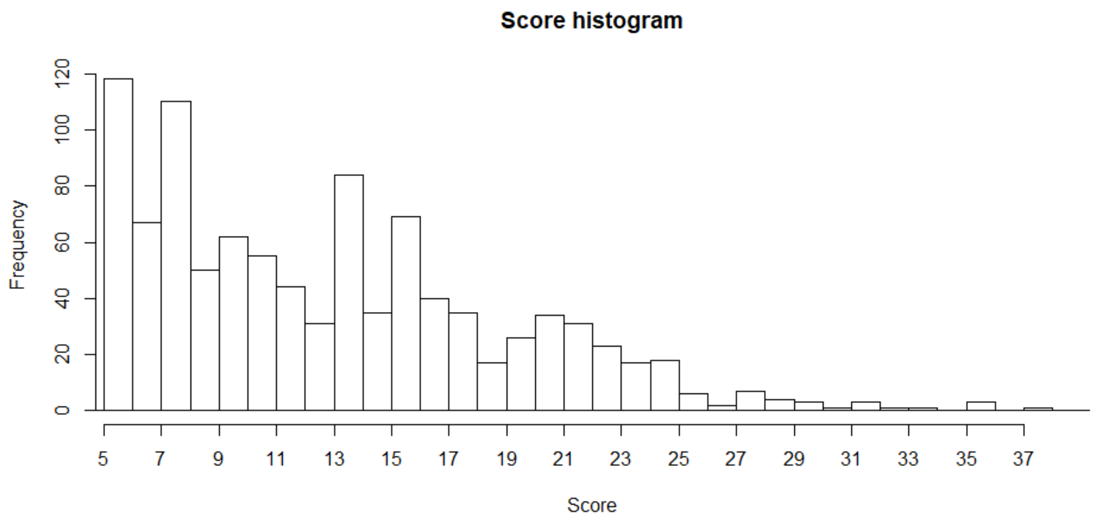

Figure 5.10: Histogram of test scores

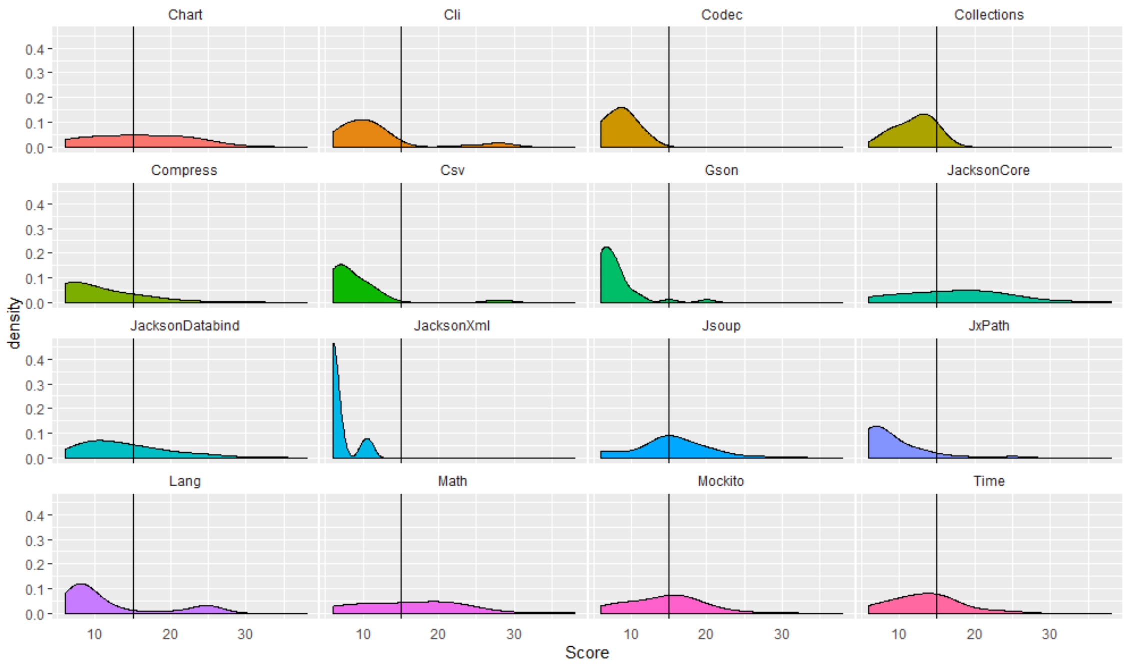

Figure 5.11: Score density per project

one trigger test, which was either low or high level. The information about the percentage of low level and high level tests from the total of triggered tests was discussed in the previous paragraphs. The focus now is on the second hypothesis: *System testing can discover the same bugs as unit testing.* To be able to give a response, an analysis has to be performed on the 153 bugs and their triggered tests, to see what percentage of bugs can be discovered by both levels. It was found that from the total of 153 bugs, 88 bugs were discovered only by low level tests, 25 bugs were discovered by high level tests and 40 bugs were discovered by both levels of tests. The percentage amounts to 57.52% for low level, 16.34% for high level and 26.14% for both levels of tests. In this case, only having the percentage is not enough to make a clear judgment, because different bugs have different number of triggered tests associated with it. The minimum number of triggered tests for a bug is 2 and the maximum number of triggering tests is 27. Because the number of triggered tests for the bugs vary, there is a difference in the degree of uncovering the bugs, as one bug that is discovered by 1 low level test and 1 high level tests would weight differently than a bug that is discovered by 1 low level tests and 26 high level tests. The difference between low level and high level regarding the degree of uncovering bugs was plotted in Fig. 5.12. The figure was made by plotting the difference between the number of low level trigger tests and the high level trigger tests, so the right side of the value 0 on the x axis represents the bugs that had more low level trigger tests than high level and the left side of the value 0 represents the bugs that were uncovered by more high level tests than low level. From the 40 bugs that had multiple triggered tests, 20 had a proportion of 50-50% of low and high level triggered tests, 10 had more low level trigger tests than high level and 10 had more high level trigger tests than low level. The difference stays close to the 0 value, with the exception of two bugs, where the number of high level trigger tests exceeds the number of low level trigger tests by 23 and 11 respectively. Fig. 5.13 shows the percentage of low level triggered tests and high level triggered tests for the bugs discovered by both levels. As can be seen from the figure, the percentages do not favor either direction. It should be noted that the amount of data points for this analysis is not extensive, so we cannot draw a strong conclusion.

Considering this and the previous findings, the results suggest that system tests do not discover the same bugs as unit tests. This fact can also be attributed to the fact that there are more low level tests than high level tests in the sample.

Until this point we discussed the fault detection power based on the level of tests and we analysed the bugs that were uncovered by multiple tests to see if one level is more favorable in detecting them. But due to the construction of the framework, we have information about the specific metrics that form the framework for each test, which is linked with the number of bugs that the specific test uncovered. An interesting aspect to investigate is how each metric correlates with bug detection and possibly with the level of test. Graphs were made to determine if there is any interesting trend in data regarding these relationships. For each of these metrics: number of methods, classes and packages in test body, number of methods, classes and packages in code tested and the condition of a big setup, two graphs were build, one for the relationship between the metric and the condition of uncovering bugs

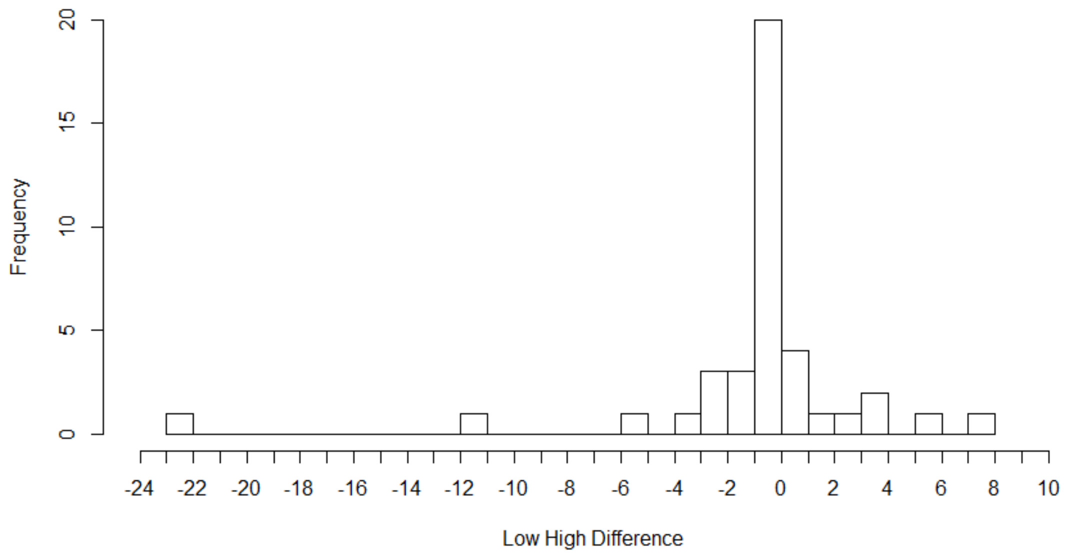

Figure 5.12: Low-High tests difference for bugs

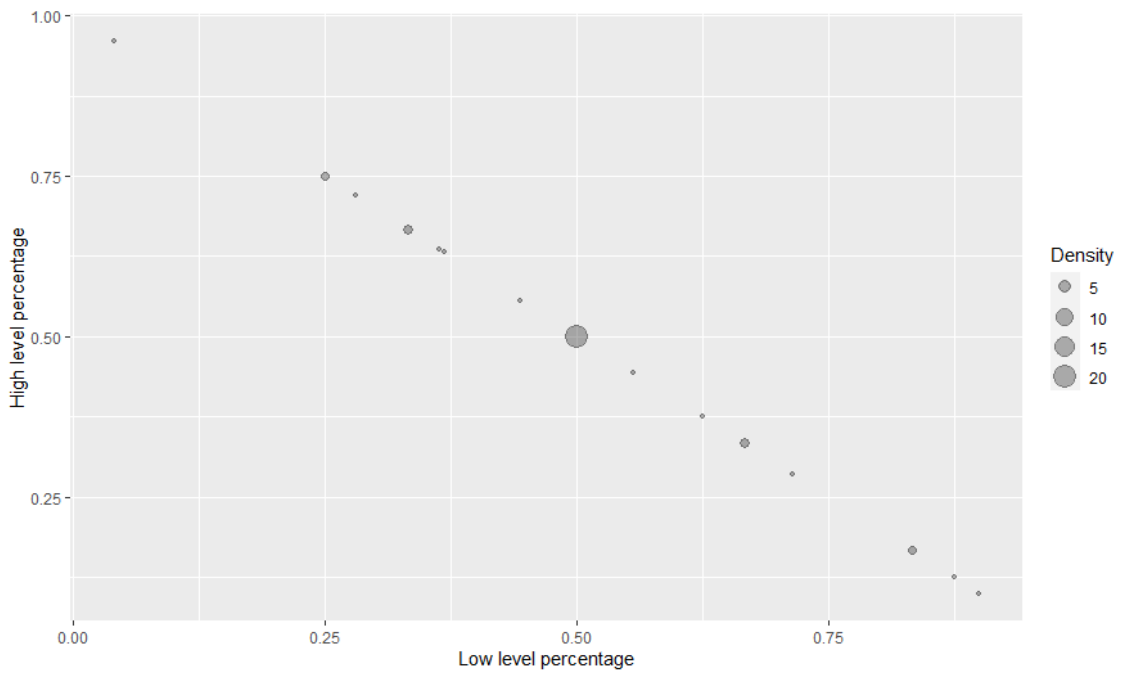

Figure 5.13: Low-High tests difference for bugs

and the second one conveys the relationship between the metric, the condition of uncovering bugs and the level of test. The use of mocking as a metric was excluded from the analysis as the number of test that employ mocking in their implementation is insignificant compared to the total number of analysed tests. Because of this aspect, we could drop the metric from the framework, as it would not make any difference in the testing levels for the tests in our sample. Regardless of this fact, the metric will remain part of the framework, but should be further analysed in the context of Java open source projects that contain mocking, to see if the relationship between the metric and the level is a strong one. For the size of setup characteristic, the analysis was performed only on the condition of having a big setup, not independently on the three metrics (number of methods, classes and packages), because that is the condition for awarding the points for the total score of the test.

In Fig. 5.14 we can see the tests being plotted with their respective number of methods over the condition of uncovering bugs. The Fig. 5.15 shows the same information with the addition of color coding based on test levels. An outlier test was excluded from the graphs, as its number of methods had a value way larger than 1000. From the second figure it can be seen that the low level tests had a low number of methods, as expected based on the construction of the framework and the importance given to that metric.

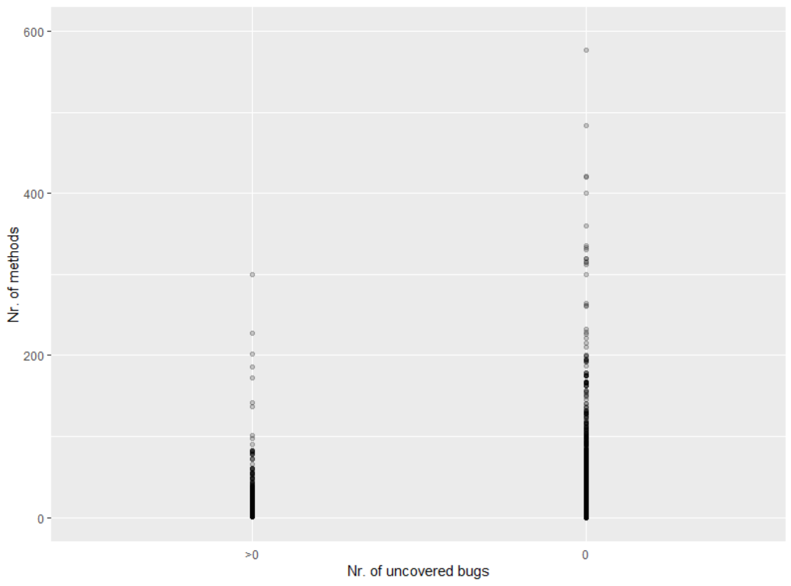

Figure 5.14: Relationship between no. of methods in test and bugs uncovered

The Fig. 5.16 and 5.17 shows the relationship between the number of methods counted from the code tested and the uncovering bugs condition for tests, and for the latter figure the relationship between the two previously mentioned aspects and the level of tests. The same outlier was excluded from these graphs as well. It can be seen that the number of methods here are larger than in the previous 2 graphs 5.14, 5.15 but that is expected given the fact that this metric envelops the previous one. As expected, low level tests tend to have lower number of methods in the code tested.

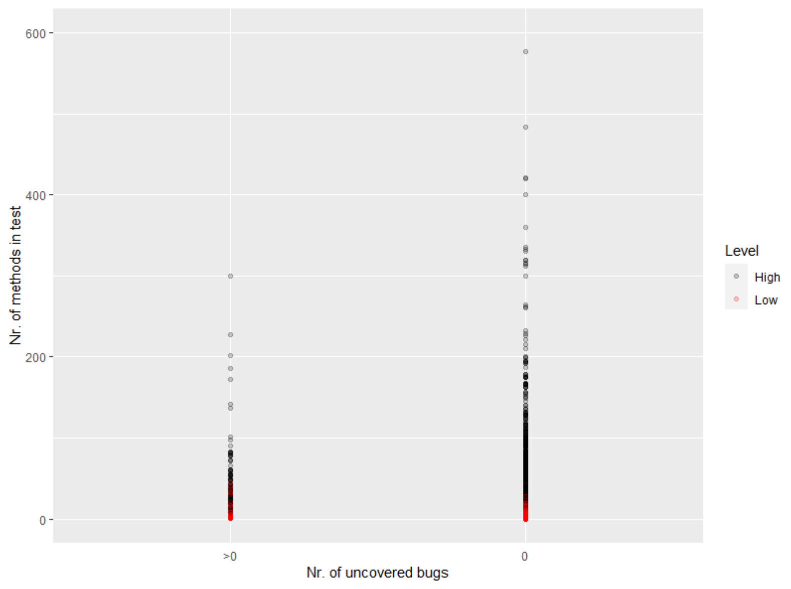

Figure 5.15: Relationship between no. of methods in test, bugs uncovered and testing level

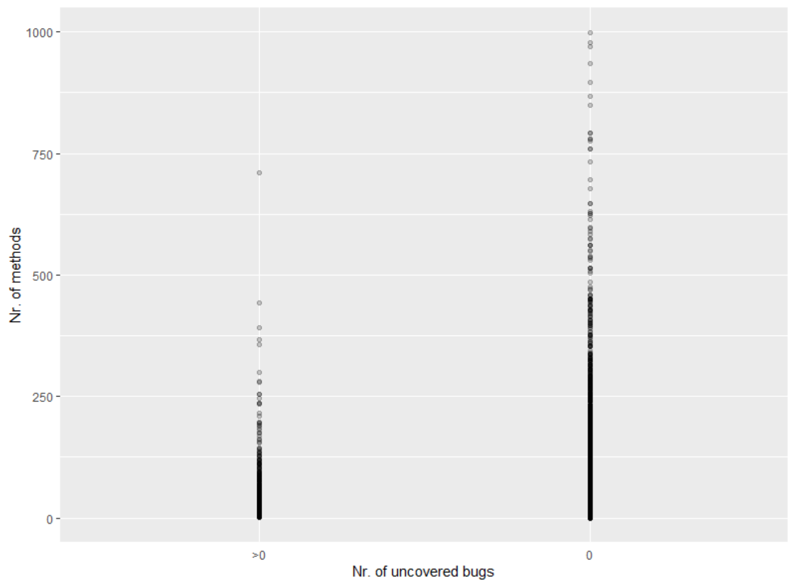

Figure 5.16: Relationship between no. of methods in code tested and bugs uncovered

Fig. 5.18 illustrates the relationship between number of classes reached from the test and bugs uncovered, while 5.19 add on top on that information the relationship with the test levels. The maximum number of tests for a specific number of classes reached was 8455, for a number of one class reached. The maximum number of classes reached was 14 for tests that did not uncover bugs and 10 for the ones that did uncover bugs. With the exception of tests that did not reference any class, with the increase of classes reached the number of tests decrease. From the tests that did

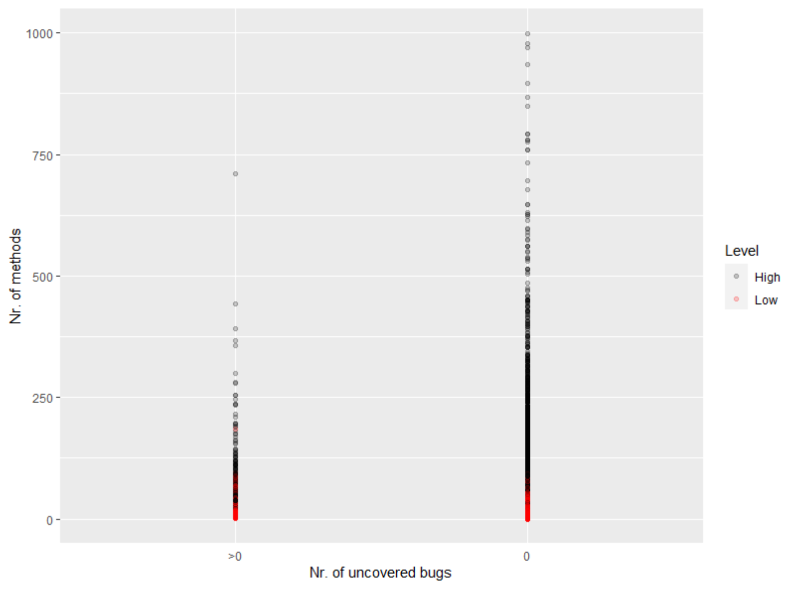

Figure 5.17: Relationship between no. of methods in code tested, bugs uncovered and testing level

not uncover bugs, the ones that had over 6 classes were all identified as high level, with the amount of 433. From the tests that uncovered bugs, the ones that had equal or more than 6 classes referenced were all high level, which amounted to 103.

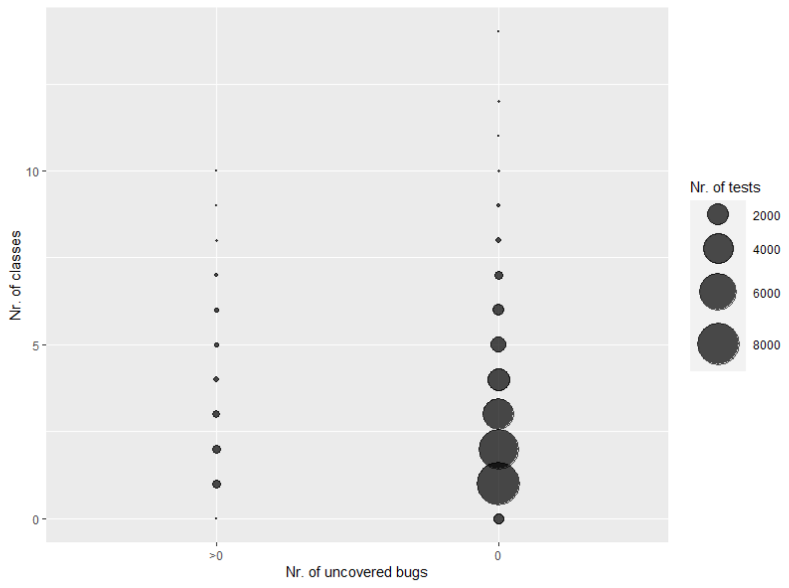

Figure 5.18: Relationship between no. of classes in test and bugs uncovered

Fig. 5.20 illustrates the relationship between no. of classes reached from the code tested and bugs uncovered, while 5.21 add on top on that information the relationship with the test levels. The maximum number of classes reached from the code tested is 29, achieved by 1 test. For this metric, the maximum number of tests for a specific class value was 4807, for tests that reached 2 classes. For the tests that uncovered bugs, the ones with over 11 classes reached were all categorized as high

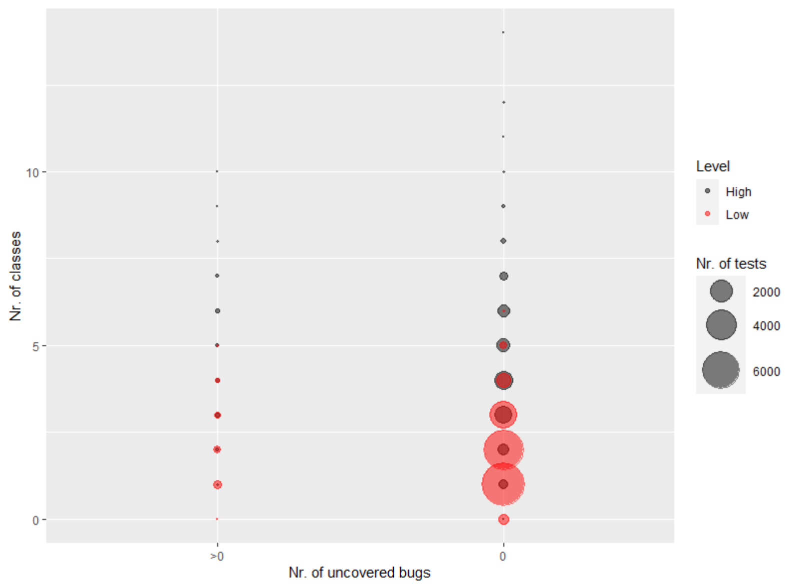

Figure 5.19: Relationship between no. of classes in test, bugs uncovered and test level

level, measuring 102 tests. From the tests that did not uncover bugs, the 103 tests that had over 13 classes were identified as high level, with the exception of 6 classes that had 17 classes reached but were identified as low level tests.

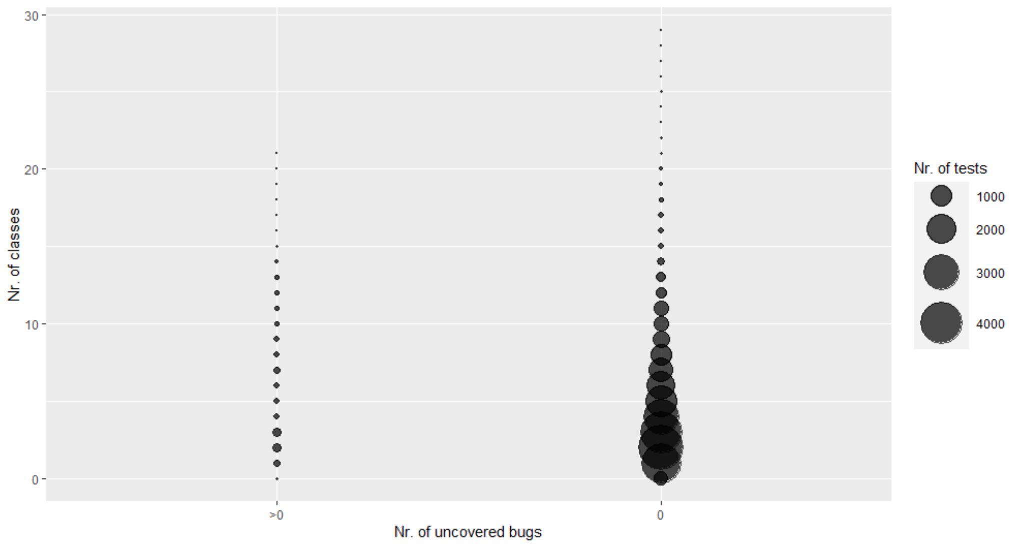

Figure 5.20: Relationship between no. of classes in code tested and bugs uncovered

Fig. 5.22 illustrates the relationship between the number of packages reached from the body of the test with the bugs uncovered and for 5.23 the relationship between the metric mentioned and the level of the tests. The maximum number of packages reached in the body of the test was 7, with 13 tests having this value. 13,905 tests reference exactly 1 package in their body, where 492 of them uncovered bugs. 1 was the most number of packages reached in a test, both for test that uncovered bugs and test that did not uncover any bugs. All the tests that had reference more than

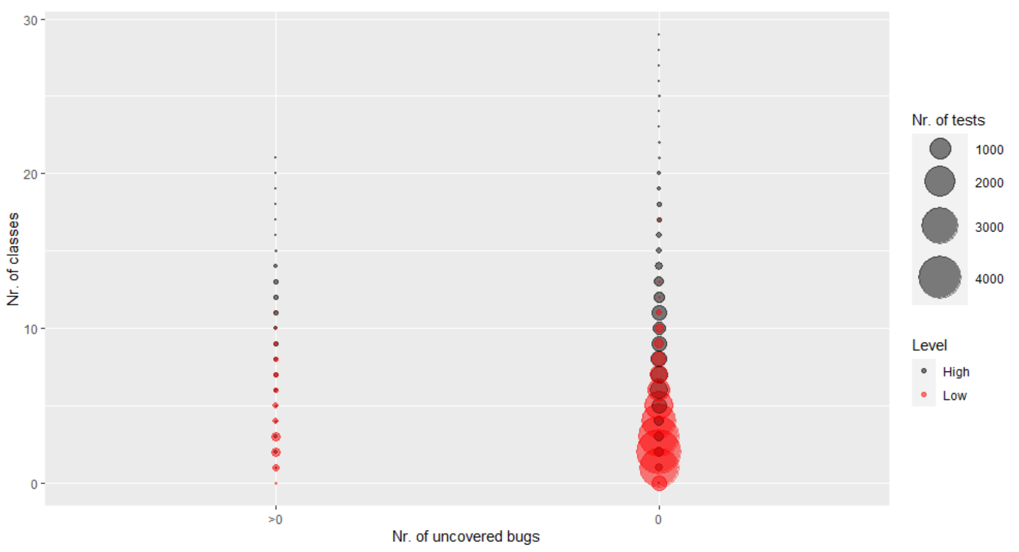

Figure 5.21: Relationship between no. of classes in code tested, bugs uncovered and test level

4 packages, which amounted to 442, were identified as high level tests. All 114 tests that uncovered bugs and had more than 3 packages reached were all high level.
A number of 5 packages reference in the test would amount to a minimum of 11 point for the size of test characteristic (5 point from number. of packages, 5 points from number of classes as the number of classes can not be less than the number pf packages, and 1 point from number of methods, as number of methods can not be less than the number of classes). Because the size of code characteristic include the size of test, the values for number of methods, classes and packages for the size of test characteristic will always be less or equal with the values for number of methods, classes and packages for the size of code tested characteristic. Hence, the minimum points for the size of code characteristic would be 8 points (4 from no. of packages, 3 from no. of classes and 1 from no. of methods), so the total score for the test would always be over 15, hence the test would be identified as high level.

Fig. 5.24 show the relationship between the packages reached from the code tested and the bugs uncovered, while Fig. 5.25 makes a distinction in that relationship based on the level of test. The maximum number of packages reached for the size of the code tested was 17, with 1 test having this value. 7766 tests reference exactly 1 package in the context of the code tested. This value is the highest number of tests for any value the number of packages takes. From the tests that did not uncover bugs, those that had over 10 package references were identified as high level, and represent 71 tests. From the tests that did uncovered bugs, those that had over 7 packages references were identified as high level, representing a total number of 43 tests.
For a sufficiently higher number of packages, the level of test would always be high. A test that will reference over 20 packages would be high level. For a test that reference between 10 and 20 packages, there is a way higher chance that it would be identified as a high level. This relationship is modeled in the framework, as if you

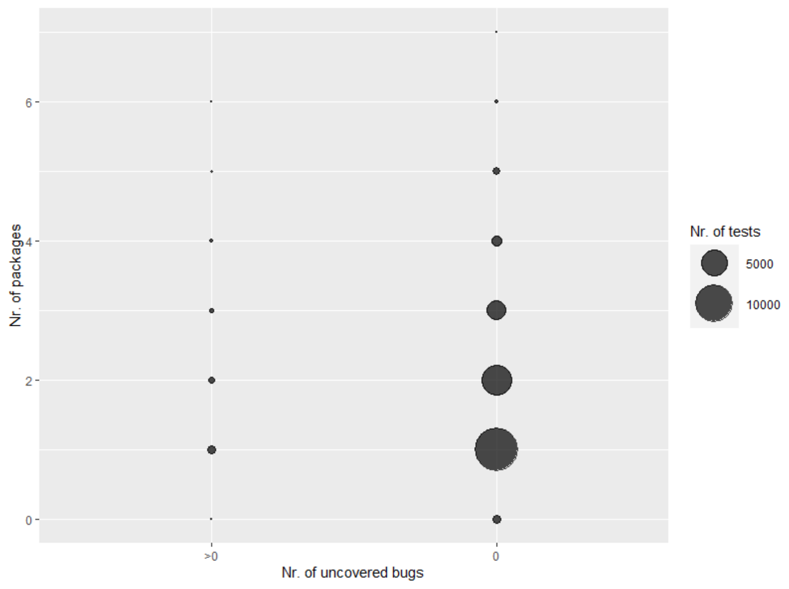

Figure 5.22: Relationship between no. of packages in test and bugs uncovered

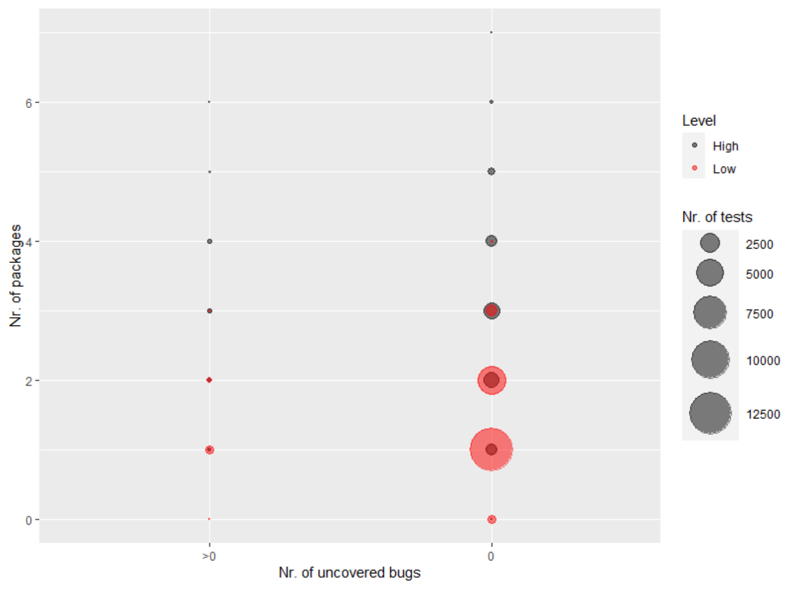

Figure 5.23: Relationship between no. of packages in test, bugs uncovered and test level

have a high number of packages, you must have an equal or higher number of classes reached, as the class can belong to only one package. Having a higher number of classes implies an equal or higher number of methods. So working on the example of a test having between 10 and 20 reference packages, it would mean that for the size of code tested, the score would be 11 points. The border between low and high level is set at 15 points, which limits the amount of points gained from size of test, setup and mock usage to 4. The smallest score for the size of test characteristic is 3, so if the number of packages reference in the test would be over 2, the test will be identified as high level.

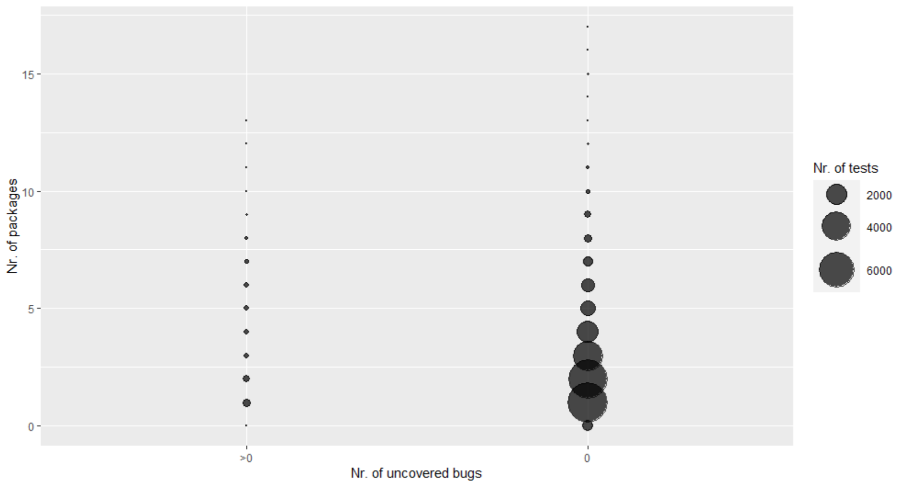

Figure 5.24: Relationship between no. of packages in code tested and bugs uncovered

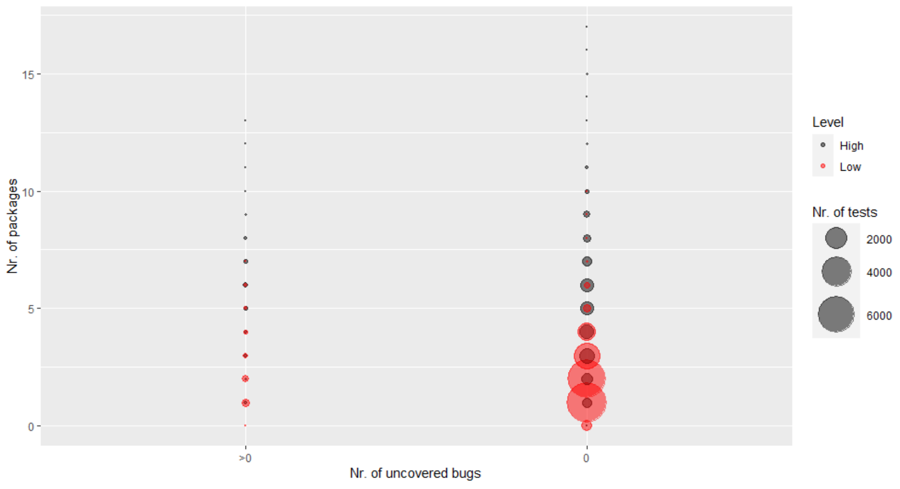

Figure 5.25: Relationship between no. of packages in code tested, bugs uncovered and test level

As seen in the Fig. 5.26 and 5.27, from the total of 25477 tests, 3835 had a big setup phase. From those 3835 tests with big setup phase, only 65 uncovered bugs, from which 42 tests were identified as low level and only 23 as high level. The rest of 933 tests that uncovered bugs either had a small setup or no setup at all. From the total of 998 tests that uncovered bugs, the rate of tests with big setup is approx. 6.51%.

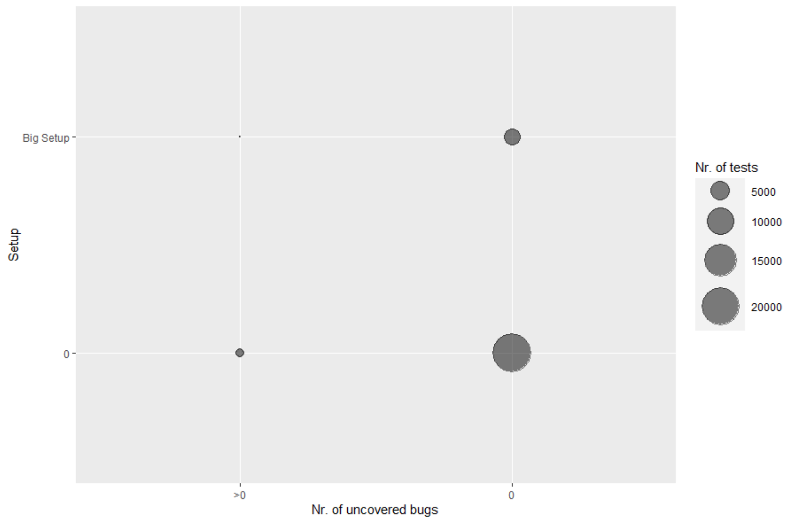

Figure 5.26: Relationship between big setup and bugs uncovered

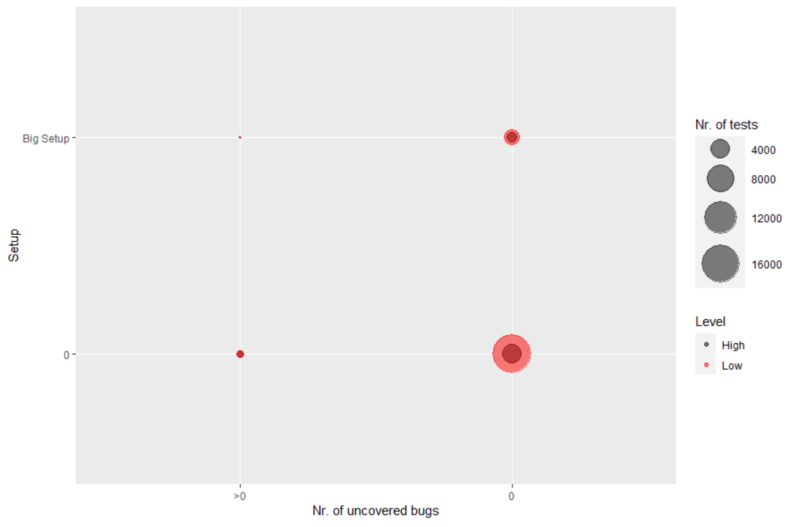

Figure 5.27: Relationship between big setup, bugs uncovered and level of test

For the number of classes and packages metrics associated with the size of test and size of code tested, is seems that an increase in the value of the metric means a decrease in the number of tests that have that value for the metric. Another observation that can be made from the graphs is that tests on the low level have a lower value for the metrics, but that is due to the construction of the framework and it was expected.

Looking at the data conveyed in the relationship graphs, there seems to be no discerning trend between any particular metric and bug finding.

5.4 Projects Information

In this section, some aspects of testing process and efforts of the selected projects will be discussed. Some of the information presented in this chapter comes from the data gathered for the categorization framework and some information comes from independently analysing the projects. Although this information is not linked directly to the research questions, it can provide insight to other aspects of testing. 16 individual projects were analysed, but between those projects there are two clusters of projects that are developed under the same umbrella: Apache [44] and FasterXML [45]. Three projects are under FasterXMl: jackson-core and jackson-databind which are core modules of the Jackson project and jackson-dataformat-xml which is an extention component. The projects under Apache are commons-cli, commons-codec, commons-collections, commons-compress, commons-csv, commons-jxpath, commons-lang, commons-math, in total 8 projects. The remaining 5 projects are independent: jsoup, jfreechart, gson, mockito and joda-time. This is specified because the testing process can be similar for projects that belong to the same cluster, regardless of the open source nature of the projects.

There were in total 25477 tests, from 2940 classes contained in 352 packages. It should be mentioned that only the active tests were counted, as well as the classes that had at least one active test and did not fulfill the excluding criteria mentioned in Chapter 3.1.3. Likewise, only packages that contained at least one active class were counted. The number of tests per projects varies from 235 to 4248, while the number of test classes varies from 20 to 557. The number of packages varies between 2 and 88. The project that has the most number of tests is Time, containing 4248 from 127 test classes contained in 7 packages, while the project with the smallest amount of tests is Cli, with 235 tests from 25 classes contained in 2 packages. The concrete information for each specific project can be seen in table 5.3.

Through the data collection process, information was obtained about the usage of mock libraries in the projects tests. From the total of 25477 analysed tests, only 63 tests employed the help of a mock library in their implementation, which represents 0.25% of total tests. From the 16 projects, only 5 contained tests with mock usage. The mocking libraries used are easymock [46], mockito [47], powermock [48] and mockrunner [49]. Mockito and easymock were the two libraries most used throughout the projects. It is necessary to mention that one of the analysed projects is in fact a mocking library - Mockito, so for this project the use of the project itself was not registered as mock usage as in fact the mocking implementation was the one that is tested. This information per project is summarised in the table 5.4.

Data about the usage of a setup phase was collected as part of the categorization framework. From the 16 analysed projects, only one does not employ any form of setup for any of the test classes contained in the project, while the other 15 employ it for a number of the test classes. From the total of 2940 test classes, only 384

Table 5.3: Projects size regarding tests

Project	**No. of Tests**	**No. of Classes**	**No. of Packages**
Chart	2244	341	26
Cli	235	25	2
Codec	842	66	7
Collections	1680	204	21
Compress	1250	147	27
Csv	343	20	3
Gson	1053	95	10
JacksonCore	729	141	14
JacksonDatabind	2652	557	44
JacksonXml	305	113	14
Jsoup	862	43	7
JxPath	331	41	14
Lang	3320	198	14
Math	3168	358	54
Mockito	2215	464	88
Time	4248	127	7

Table 5.4: Mock usage

Project	**Total tests**	**Mock tests**	**% of mock tests**	**Mock libraries used**
Collections	1680	3	0.18%	easymock
Compress	1250	3	0.24%	mockito
JacksonDatabind	2652	16	0.6%	mockito, powermock
JxPath	331	4	1.2%	mockrunner
Lang	3320	37	1.1%	easymock

of them contain a setup phase, which represents approx. 13% of the total test classes. The number of classes with no setup, number of small and big setup phases were recorded per project and are shown in the Fig. 5.28.The JacksonCore project had no test classes that contained setup phases. The next project with the least amount of setup phases was JacksonDatabind, where from 557 test classes only 2 had setup phases, which represents 0.36% from total test classes. The project with the highest percent of setup phases is Time, where from the total of 127 classes, 95 had setup phases, amounting to 74.8%. Returning to the size condition for the setup phase established for the setup score of a test, from the 384 test classes with setup phases, 146 of them are considered big regarding size, which represents approx. 38%. Because JacksonCore project did not have any classes containing setup, the percent of big setup phases is also 0. The next project with the lowest percent of big setup phases is Gson, where from a total of 41 classes containing setup phases only 1 was considered a big, representing a 2.43%. The highest percentage of big setup phases was 75%, registered by the JxPath project, where from the 16 test classes with setup phases 12 of the setup phases were regarded as big.

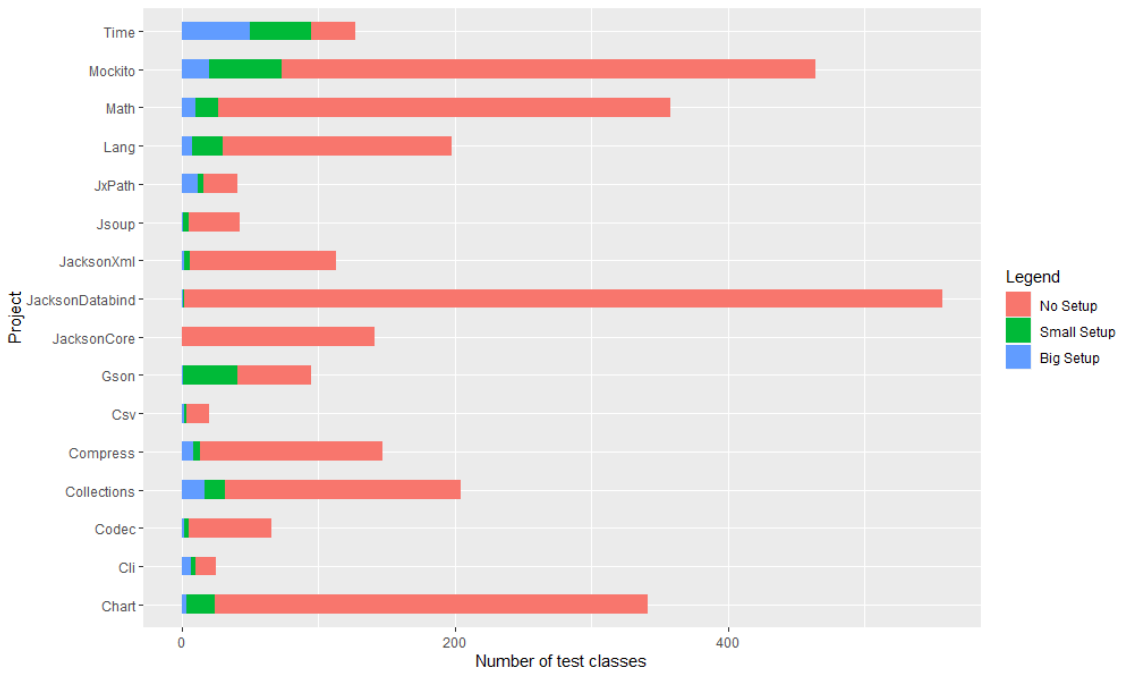

Figure 5.28: Setup state per project

6
Discussion

In this chapter we discuss our study in terms of implication to both practitioners and research, limitation of the study and validity concerns.
The concrete contributions of the study are a framework to classify on which level a test resides followed by an assessment of which testing level is used more in Java open source projects, as well as which testing level can uncover more bugs. The two assessments are based on the test categorization made by the developed framework. The framework shows that there are metrics that can act as proxy for the level of test. Our findings suggest that low level /unit testing is the level of testing used more in Java open source projects. Additionally, although the low level uncovered more bugs than high level, considering the percentage of low and high level tests in the sample, the high level had a higher detection power. Regarding bugs, the findings suggest that high level tests do not detect the same bugs as low level tests.

The results for the research questions two and three, which ask about which level of testing is more used in Java open source projects and what level has a higher rate of uncovering bugs, are based on the categorization made by the framework developed as a response for research question 1, which inquires about metrics that can act as an identifier for the level of tests. The validity of the framework is important because all the results are based on it, so if the framework is not constructed in an adequate manner, then our findings are not valid. Moreover, if the framework contains bias to one of the levels, the results would be skewed. A verification of the framework was performed, and through it some limitations were outlined. These limitations are discussed in the section 6.1.

To determine which level of testing is most used, the framework was applied on the tests and then we counted the tests that were identified on each level, to find that low level tests are more in number. Moreover, this result was valid on the total number of tests as well as per each project. A possible explanation for the findings can be the fact that from the sample projects, multiple ones are self-identified as libraries, so it can be expected that they would contain more low level testing than high level ones. Other type of projects can favor other test level, possibly yielding different results. Another aspect that can influence the result is in fact the categorization framework. If the threshold between the low level and high level is too high, then it is possible that more tests would be identified as low level. Although a verification was made on the framework, there is still the possibility that the framework is biased towards the low level.

The results regarding the fault detection power of the two testing levels can be interpreted in two ways. First, if we only focus on the absolute numbers, the low level had significantly more tests that uncovered bugs than the high level, but this outcome can be attributed to the fact that there was a higher number of low level tests in the sample. In fact, the number of high level tests was less than a third of the number of low level tests, while the number of high level tests that uncovered bugs was more than half of the number of low level tests that uncovered bugs. The difference between the total number of tests per level is considerable, but we can not say the same about the number of tests uncovering bugs per level, as the total number is 998 tests, which amounts to 3.92% of the total number of tests in our sample. Another interpretation can be made if we look at the rate of uncovering bugs. For the low level, this rate was at 3.28% while the rate of uncovering bugs for high level was at 6.23%. Looking only at this rate, the high level performed better than low level. This outcome can be influenced by the fact that the total number of tests uncovering bugs is low compared with the total number of tests. For our context, the rate of uncovering bugs is a more interesting outcome.

Besides the fault detection for the levels of testing, we looked to see if both levels can find the same bugs and for bugs that were uncovered by multiple tests, which level had more triggered tests. There were 561 bugs with a total of 998 triggered tests, but only 153 had multiple triggered tests, so the number of data points for this analysis was low. The results suggest that low level uncovered more bugs that were uncovered by multiple tests. For the bugs that were uncovered by both levels, there seems to be no preferred level on which the bug would be uncovered by more tests on a level than the other. Owning to the fact that the number of data points is low, we cannot draw a strong conclusion. Once more, this result can be attributed to the fact that there are more low level tests in the sample, so there is a possibility that the low level covers more of the software than the high level. The focus was on how the numbers compare, so we cannot say the reason why the results are how they are. There are multiple possibilities regarding the reason for high level not uncovering the same bugs as low level. First, the tests may not cover the same part of the system. If the high level tests focus on a part of the system and low level tests focus on another, there would not be many interlapping. Even if the two levels of testing focus on the same portion of the software, they may test different scenarios that would bring different inputs for the code that contains the bug, where for some inputs the code functions correctly and for other inputs the code can trigger the fault. Similarly, high level tests might not be able to reach that faulty code with the full range of inputs due to limitations in the scenario tested, while low level tests that focus on that piece of faulty code can. Another possibility would be that a fault detected at a low level be masked by some higher level implementation hit by the high level test, hence it would be hidden and not registered by the high level test. From these discussed scenarios, we can discern two plausible prospects, first is that high level testing has an inherently inability in detecting bugs on low level and second is that high level testing is not done extensively to cover all the low level parts. Taking one step further, we started an analysis to determine if there is a trend in the data between the metrics in the framework and bug uncovering, but we could not find any meaningful relationship.

6.1 Limitations

The tool used for analysis limits the scope of open source candidates to projects with public repositories and issue trackers that are supported by the tool (google-code [32], jira [33], github [34] and sourceforge [35]). Another limitation brought by the tool is that the code language used in the projects should be Java.
The study focuses only on the fault detection power of unit and system tests, not examining the effect of integration tests. While we ideally would study multiple different testing techniques and not only their fault detection capabilities but also their cost etc., this is not feasible in this project. We thus focus on maybe the two most common testing levels, unit and system testing, and on the primary difference between them, i.e. their fault detection capabilities and how they compare. We argue this is a first natural step towards guiding practitioners in trading off between them. Later research will have to focus on the associated costs to complement our results.

The verifications performed on the developed framework uncovered some limitations in its capability to categorize tests. The categorization framework was developed for a general development of tests and may not be fully accurate for different approaches to testing. There are different types of testing with different good practices associated with it. The general way of testing is preparing the environment and other needed settings in the setup phase if needed, in the body of the test the input for the piece of code tested would be prepared, then it would be fed into the code and the results would be assessed against the true correct value, followed if needed by a tear-down phase where the environment and settings can be returned to the default values. There are other ways of doing testing that is different than the general layout. For example, in one of the projects there were tests that contained only one method, which was a private method in the same testing class that contained the actual test code. In this case, the framework would consider the private method in the body of the test as the code that is tested and the body of the private method the implementation of the code tested, while in reality, the actual code tested is in the private method of the test class. Another case where the framework would give skewed results is when the project developed its own assertion methods. Because the assertion methods would be considered code of the project, the framework would count the body of the assertion method for the size of code tested for a test that uses that specific assertion.
The framework would not work accordingly for tests that contain multiple test cases, but as it is a testing anti-pattern, this mistake is attributed to the developers or testers that wrote the code, not to the framework. Another restriction put on the framework because of the open-source nature of the analyzed projects is the lack of access to the requirements of the projects. If the requirements would have been accessible, the framework could have been taking advantage of it when categorizing tests on the high level. In the current formula, the framework has issues with cate-

gorizing tests that in size are small and constraint to a small number of classes, but regardless it represents a requirement for the full system. These represent limitations of the categorization framework.

6.2 Implications

Implications to practitioners

The study can help practitioners of open source development in deciding where the testing efforts should be concentrated based on the fault detection power of different levels of testing. Knowing that unit testing is the level of testing preferred can influence practitioners in their choice for adopting testing practices. Moreover, knowing that high level tests have a higher rate of fault detection means it might push practitioner into using this level more prominently throughout the testing phase. The study is focused on the fault detection power of the unit and system levels, but in taking a decision to which level of testing to implement, multiple consideration should be taken into account. The differences in cost, time and developers experience were not researched in this study, but they are factors that should weight in when taking a decision about the usage of the respective testing levels.

Implication to research

For researchers, the study supports and opens new research possibilities in the subject. Further studies can be done in determining testing efforts and return of investment of different levels of testing. Along with our study, that could be a comprehensive guide of what testing level should be chosen by practitioners based on their needs and possibilities. The current study found that unit testing is the preferred level in open source projects, but it could not make any assumptions on the reason why, so this can be a further research opportunity. Moreover, it could be interesting to compare the reason why the open source community uses more unit tests with the reason given by the close source/ proprietary software development community. The data points used to make an assessment about the fault detection power of the levels are not as extensive to allow us to assume generalizability without a doubt, as there were 998 triggered tests to be categorized and 153 bugs that were uncovered by multiple tests. Compares to the total number of tests, these amounts are very low. This research has to be repeated on a larger set of projects to be able to replicate the results. Moreover, having the results about the fault detection power of high level tests allows for research to be done on the cost, time, return of investment of high level testing and the disadvantages of high level testing, in order to have a well-rounded empirical data to help with the testing decisions. Additionally, this paper started an inquiry on mock usage in Java open source projects and what mock libraries are the most used. The data was not enough to gather a conclusion, but it can be used as initial data in a more elaborate research.
Another topic of research can be in the line of the characterization framework developed in the paper. Researcher can use the framework to determine the level of tests in their research, or build over the framework to perfect it. Our framework caters to only two out of three levels: unit and integration. Further studies can be done to determine where would the integration level be situated in the framework

and the threshold between the integration level and the other two would reside. Moreover, a more comprehensive analysis can be made of the framework, making a linear model with the metrics as predictors and determining which metric has the more significance in determining the level of tests.

6.3 Validity Consideration

Threats to internal validity
The study was done on mature projects to counteract the effects of different stages of development on testing efforts. The process of selecting the open source projects to be part of the study can introduce selection bias. To counteract this, the projects chosen would be from different domains. For the data collection we rely on the Defects4J tool. If the tool is defective it can introduce bias. Another element that can introduce bias is the framework created for classification of the levels of testing. It should be noted that the way the development process is dealing with bug reports and replication can skew the results towards a level or another. This process follows the following steps: a bug is reported, tests are designed to reproduce the respective bug, fixes are made for the issue and the test check the fixed issue. Another possibility would be that the code associated with a bug did not have tests, and the tests had been added later then running the tests the bug is discovered.

Threats to external validity
External validity is concerned with how the findings of a study can be generalized in a broader context. For the study, 16 Java open source projects were selected. It should be possible to generalize the findings based on the number of projects selected. The results can vary if other projects are chosen, thus replication with other projects is needed for the conclusion to be generalized.
In the study, only open source Java projects were selected. We acknowledge that the results cannot be generalized for other programming languages. Although the framework for categorizing tests is targeted at the Java language, it can be adapted to fit different programming languages that contain OOP principles. As a result of using a definition for "unit" that is targeted to OOP paradigms, the framework cannot be applied on different paradigms while expecting accurate results.
Due to the fact that the study analysed open source projects, the results cannot be generalized to traditional closed source projects. There are multiple differences between the development activities of the open-source software and closed-source software, such as requirements elicitation, documentation, release strategy, as depicted by Llanos et. al. [37]. Another major difference is in the testing approach, where in open-source software development users take the role of bug reporters, bringing notice of the bug to the community or try to solve it as anyone can contribute to the OS code, while traditional closed-source software development uses service packs to repair bugs [38]. The vast community of users of an open-source software acts as testers, where closed-source software have dedicated teams for testing. There are as well differences in the testing approach between them. Considering the fact that either one of the differences between the two approaches can influence the level of testing used, we cannot generalize the findings to traditional closed-source software.

The target population was defined as Java open source projects containing test suits. One problem with the selection of the sample could be that the selected projects do not have both levels of testing that are important for the study. This would still help the study respond to the RQ2, but would not be as helpful for RQ3. By selecting only projects that contain both levels of testing, selection bias can be introduced, as there could be a preference of using one level of testing over another in the open source development. Another problem with this approach would be the fact that the selection would be based on a specific definition of unit and system testing, which would be later in the study challenged, as one of the research question of the study is finding metrics to act as a proxy for testing levels.
Due to the nature of the study there should be no experiment or testing effects to threaten the external validity of the study.

Reliability
Reliability is concerned with the degree to which the findings can be reproduced when the research is repeated under the same conditions. The study can be replicated, as the investigated projects are available on online open code hosting platforms, with public issue trackers and Defects4J is available on GitHub. The SequenceDiagram plugin is also available in the Intellij Plugin Marketplace. For reliability we included information about the selection of projects, the method of collecting data and data exclusion constraints. The data collection process is thoroughly described and the developed framework for characterization of tests presents clear metrics to measure.

7 Conclusion

7.1 Conclusion

In this paper we analysed Java open source projects to investigate the state of testing in real world projects. In particular, we examine the projects to determine which level of testing is more prevalent and how the fault detection power of the respective levels compares to each other. Only unit tests and system tests were considered. Next, we proposed a framework to categorize tests that is based on concrete and measurable metrics. The data was collected from the projects' repositories and aided by the Defects4J tool. The framework was applied to the testing suites of the selected projects, followed with an analysis on the fault detection behavior of the test suites. A further analysis was conducted in respect to the research questions of the study.
The categorization framework proposed is based on a score system applied to the size and isolation characteristics. Mock usage was considered for the isolation characteristic. For the size characteristic, the number of methods, classes and packages were the chosen metrics and they were applied for the size of test, the size of code tested and the size of the setup. Based on the values of the metrics, a score was assigned for each metric, and the sum of all the score metrics become the total test score which would be compared to a threshold value to determine the level for the respective test.
The most prevalent level of testing was found to be low level testing, with a proportion of 78.4% of the total tests. Considering the detection power of the levels of testing, 65.63% of the tests that uncovered bugs were low level tests. If we consider the tests that uncovered bugs from the total of tests on the same level, the low level achieved a rate of discovery of 3.28% while the high level achieved a rate of 6.23%. We can conclude that high level tests have a greater fault detection power than low level tests.

7.2 Future work

For future work, the study should be replicated using different projects. Moreover, the study can be replicated using a different bug database, such as Bugs.jar, BugSwarm and Bears, which are bugs data-sets for Java. Another interesting direction for future work would be investigating all the revisions of the projects that relate to bug fixes (so the revision with the bug and the revision that fixed the bug), as this could provide an insight into the evolution of testing. Another step would be to adapt the categorization framework to be fully language independent for programming languages with object-oriented principles. Moreover, the framework should be extended for other programming paradigms. Additionally, we should investigate the level of integration testing in open source Java projects, how the usage of integration tests compares with the usage of unit and system tests in open source projects and how the fault detection power of integration tests compares to the fault finding behaviour of unit and system tests. Furthermore, a study on the fault detection power of different levels of testing should be done in a traditional closed-source context. This would provide information on the current state of testing practices in the industry.

Bibliography

[1] 14th Annual State of Agile Survey Report VersionOne, 2020, [online] Available: http://stateofagile.versionone.com.

[2] Madeyski, L. and Szała, Ł., 2007, September. *The impact of test-driven development on software development productivity—an empirical study.* In European Conference on Software Process Improvement (pp. 200-211). Springer, Berlin, Heidelberg.

[3] Hellmann, T.D., Chokshi, A., Abad, Z.S.H., Pratte, S. and Maurer, F., 2013, August. *Agile testing: a systematic mapping across three conferences: understanding agile testing in the xp/agile universe, agile, and xp conferences.* In 2013 Agile Conference (pp. 32-41). IEEE.

[4] Jones, C., 1998. *Applied software measurement assuring productivity and quality.*

[5] Ali, I., 2009. *A comparison between Black Box and White Box Testing.* Interesting Results in Computer Science and Engineering.

[6] Offutt, J., Pan, J. and Voas, J.M., 1995, June. *Procedures for reducing the size of coverage-based test sets.* In Proceedings of the 12th International Conference on Testing Computer Software (pp. 111-123). New York: ACM Press.

[7] Vocke, Ham. *The Practical Test Pyramid (2018)*, https://martinfowler.com/articles/practical-test-pyramid.html

[8] ISO/IEC/IEEE 24765, 2010. 3.2758: Systems and Software Engineering–Vocabulary.

[9] https://glossary.istqb.org/en/search/

[10] Trautsch, F., Herbold, S. and Grabowski, J., 2020. *Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects.* Journal of Systems and Software, 159, p.110421.

[11] Xie, T., Taneja, K., Kale, S. and Marinov, D., 2007, May. *Towards a framework for differential unit testing of object-oriented programs.* In Second International Workshop on Automation of Software Test (AST'07) (pp. 5-5). IEEE.

[12] Trautsch, F. and Grabowski, J., 2017, March. *Are there any unit tests? an empirical study on unit testing in open source python projects.* In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST) (pp. 207-218). IEEE.

[13] Whittaker, J.A., 2000. *What is software testing? And why is it so hard?.* IEEE software, 17(1), pp.70-79.

[14] Runeson, P., 2006. *A survey of unit testing practices.* IEEE software, 23(4), pp.22-29.

[15] Kochhar, P.S., Bissyandé, T.F., Lo, D. and Jiang, L., 2013, July. *An empirical study of adoption of software testing in open source projects.* In 2013 13th International Conference on Quality Software (pp. 103-112). IEEE.

[16] Kochhar, P.S., Thung, F., Lo, D. and Lawall, J., 2014, December. *An empirical study on the adequacy of testing in open source projects.* In 2014 21st Asia-Pacific Software Engineering Conference (Vol. 1, pp. 215-222). IEEE.

[17] Farooq, S.U. and Quadri, S.M.K., 2012. *Quality practices in open source software development affecting quality dimensions.* Trends in Information Management (TRIM), 7(2).

[18] Aberdour, M., 2007. *Achieving quality in open-source software.* IEEE software, 24(1), pp.58-64.

[19] Basili, V.R. and Selby, R.W., 1987. *Comparing the effectiveness of software testing strategies.* IEEE transactions on software engineering, (12), pp.1278-1296.

[20] Frankl, P.G. and Iakounenko, O., 1998, November. *Further empirical studies of test effectiveness.* In Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering (pp. 153-162).

[21] Frankl, P.G. and Weiss, S.N., 1993. *An experimental comparison of the effectiveness of branch testing and data flow testing.* IEEE Transactions on Software Engineering, 19(8), pp.774-787.

[22] Briand, L. and Labiche, Y., 2004. *Empirical studies of software testing techniques: Challenges, practical strategies, and future research.* ACM SIGSOFT Software Engineering Notes, 29(5), pp.1-3.

[23] Elbaum, S., Chin, H.N., Dwyer, M.B. and Jorde, M., 2008. *Carving and replaying differential unit test cases from system test cases.* IEEE Transactions on Software Engineering, 35(1), pp.29-45.

[24] Runeson, P., Andersson, C., Thelin, T., Andrews, A. and Berling, T., 2006. *What do we know about defect detection methods?[software testing].* IEEE software, 23(3), pp.82-90.

[25] Berling, T. and Thelin, T., 2004, September. *An industrial case study of the verification and validation activities.* In Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717) (pp. 226-238). IEEE.

[26] Just, R., Jalali, D. and Ernst, M.D., 2014, July. *Defects4J: A database of existing faults to enable controlled testing studies for Java programs.* In Proceedings of the 2014 International Symposium on Software Testing and Analysis (pp. 437-440).

[27] DeMillo, R.A., Lipton, R.J. and Sayward, F.G., 1978. *Hints on test data selection: Help for the practicing programmer.* Computer, 11(4), pp.34-41.

[28] https://pitest.org/

[29] Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N. and Le Traon, Y., 2018. *How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults.* Empirical Software Engineering, 23(4), pp.2426-2463.

[30] Rani, S., Suri, B. and Khatri, S.K., 2015, September. *Experimental comparison of automated mutation testing tools for java.* In 2015 4th International Confer-

ence on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions) (pp. 1-6). IEEE.

[31] Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E. and Malevris, N., 2016, October. *Analysing and comparing the effectiveness of mutation testing tools: A manual study.* In 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 147-156). IEEE.

[32] https://code.google.com/

[33] https://www.atlassian.com/software/jira

[34] https://github.com/

[35] https://sourceforge.net/

[36] http://vanco.github.io/SequencePlugin/

[37] Llanos, J.W.C. and Castillo, S.T.A., 2012, June. *Differences between traditional and open source development activities.* In International Conference on Product Focused Software Process Improvement (pp. 131-144). Springer, Berlin, Heidelberg.

[38] Potdar, V. and Chang, E., 2004, May. *Open source and closed source software development methodologies.* In 26th International Conference on Software Engineering (pp. 105-109).

[39] Kanstrén, T., 2008. *Towards a deeper understanding of test coverage.* Journal of Software Maintenance and Evolution: Research and Practice, 20(1), pp.59-76.

[40] Rothermel, G., Elbaum, S., Malishevsky, A.G., Kallakuri, P. and Qiu, X., 2004. *On test suite composition and cost-effective regression testing.* ACM Transactions on Software Engineering and Methodology (TOSEM), 13(3), pp.277-331.

[41] Orellana, G., Laghari, G., Murgia, A. and Demeyer, S., 2017, May. *On the differences between unit and integration testing in the travistorrent dataset.* In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) (pp. 451-454). IEEE.

[42] Meszaros, G., 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.

[43] https://maven.apache.org/

[44] https://www.apache.org/

[45] http://fasterxml.com/

[46] https://easymock.org/

[47] https://site.mockito.org/

[48] https://powermock.github.io/

[49] https://mockrunner.github.io/

[50] https://github.com/rjust/defects4j

[51] https://bazel.build/

A
Appendix 1

The raw data was published on Zendo, with the DOI 10.5281/zenodo.4959837 and can be found at this link: https://doi.org/10.5281/zenodo.4959837 .

Table A.1: Projects version

Project	**Commit number**	**Commit date**
Chart	ae40fd74fc4e24ca26fa337089500dd9b7be6879	21 Jan. 2021
Cli	e093df2c10db91ab24694596b95dcc05d379ba08	17 Jan. 2021
Codec	8bfca5c4228bb05e94d8027aadefaba92ff124c7	17 Jan. 2021
Collections	dc01c2583a6d77242862f92b38908d1fc28e90de	17 Jan. 2021
Compress	6fed9fb4745bd6b4506d8ddc7db23884e8bf0bdb	17 Jan. 2021
Csv	bf2f8093a49a3432be62e9fdae073e82ac78bd04	17 Feb. 2021
Gson	ada6985285ee2d1d864c77d17d9b162d78371a26	12 Oct. 2020
JacksonCore	205255fc820e45fbf2bd80cf8182707935964e04	28. Jan 2021
JacksonDatabind	5d946f2788d2be81ea9a3ec69c12db1cb9ce1dcc	28 Jan. 2021
JacksonXml	c873c76ac7ef43965ce2c284c9a95c675587ee87	27 Jan. 2021
Jsoup	73e23c1aafe283c227b22993a9c9510d1113ac86	27 Jan. 2021
JxPath	192f4c92727cf5387a8043525a1e1e1533c9ac69	26 May 2020
Lang	3e27e51770124866c530ed321f21368ea07e7740	27 Jan. 2021
Math	d71b8c93d20187da61113990f44cb16345cf72cb	17 Jan. 2021
Mockito	fbe50583ad5e66087be58d0a8ad28a079ffc72d6	27 Jan. 2021
Time	890fb7b8801f7a8132464a8d558f410373918f96	25 Jan. 2021

A.1 RQ tables

Table A.2: Framework verification

Project	**Test & Class**	**FW**	**Dev**	**Researcher**
Collections	testOnePredicateEx2 PredicateUtilsTest	8	-	low
Cli	testPrintOptionWithEmptyArgNameUsage HelpFormatterTest	11	-	low
Jsoup	nonnullAssertions HtmlTreeBuilderTest	6	-	low
Time	testPlusMonths$_int$ TestDateTime_Basics	8	low	low
Chart	testPublicCloneable DefaultWindDatasetTest	6	-	low
Chart	testPreviousStandardDateDayA DateAxisTest	36	-	high
Codec	testBase64OutputStreamByteByByte Base64OutputStreamTest	20	-	high
Mockito	should_allow_possible_argument_types ReturnsArgumentAtTest	18	-	low
Jsoup	handlesLargerContentLengthParseRead ConnectTest	19	-	high
Math	testStepSize ThreeEighthesIntegratorTest	19	-	high

Table A.3: Second framework verification

Project	Test & Class	FW	Dev	Res.
Chart	testHashcode YIntervalRendererTest	12	-	low
Chart	testGeneral HMSNumberFormatTest	7	-	low
Chart	testCloning DefaultTableXYDatasetTest	15	-	low
Chart	testDrawWithNullInfo BoxAndWhiskerRendererTest	23	-	high
Chart	testEquals MultiplePiePlotTest	28	-	low
Chart	testAdd TimePeriodValuesTest	17	-	low
Cli	testMissingRequiredGroup ParserTestCase	14	-	low
Cli	testToString OptionGroupTest	19	high	high
Codec	testUnixCryptWithHalfSalt UnixCryptTest	6	-	low
Codec	testToAsciiChars BinaryCodecTest	24	-	low
Collections	testMultiplePeek PeekingIteratorTest	11	-	low
Collections	predicatedCollection CollectionUtilsTest	10	-	low
Collections	testSwitchClosure ClosureUtilsTest	32	-	low
Collections	testPopulateMultiMap MapUtilsTest	22	-	low
Compress	multiByteReadFromMemoryConsistentlyReturns- -MinusOneAtEof Pack200TestCase	8	-	low
Compress	testDefaultExtractionViaFactory FramedSnappyTestCase	10	-	low
Compress	checkUserInformationInTarEntry TarMemoryFileSystemTest	16	-	high
Compress	testDeleteFromAndAddToZip ChangeSetTestCase	17	-	high
Csv	testProvidedHeader CSVParserTest	14	-	low
Csv	testToStringAndWithCommentMarkerTakingCharacter CSVFormatTest	26	-	low

Table A.4: Second framework verification

Project	**Test & Class**	**FW**	**Dev**	**Res.**
Gson	testGsonInstanceReusableForSerializationAnd- -Deserialization UncategorizedTest	10	high	low
Gson	testEscapedCtrlRInStringSerialization StringTest	6	high	low
Gson	testCustomNestedDeserializers CustomTypeAdaptersTest	21	high	high
Gson	testStreamingHierarchicalFollowedByNonstreaming- -Hierarchical TypeAdapterPrecedenceTest	21	-	high
JacksonCore	testSimplestWithPath JsonPointerParserFilteringTest	13	-	low
JacksonCore	testNoAutoCloseReader ParserClosingTest	23	low	low
JacksonDatabind	testSuperClass TestRootType	14	low	low
JacksonDatabind	testPathRoundTrip TestJava7Types	14	-	low
JacksonDatabind	testRequiredNonNullParam FailOnNullCreatorTest	12	-	low
JacksonDatabind	testXMLGregorianCalendarSerAndDeser MiscJavaXMLTypesReadWriteTest	17	-	high
JacksonDatabind	testKeyDeserializers TestKeyDeserializers	18	low	low
JacksonDatabind	testReaderFailOnTrailing FullStreamReadTest	24	-	high
JacksonXml	testSimpleKeyMapSimpleAnnotation Issue37AdapterTest	7	low	low
JacksonXml	testPolyIdList178 JAXBObjectId170Test	16	low	low
Jsoup	handlesUnknownInlineTags HtmlParserTest	15	-	low
Jsoup	descendant SelectorTest	24	-	high
JxPath	testCreatePathAndSetValueDeclVar- -SetCollectionElementProperty VariableTest	8	-	low
JxPath	testDoStepNoPredicatesPropertyOwner SimplePathInterpreterTest	24	-	low

Table A.5: Second framework verification

Project	**Test & Class**	**FW**	**Dev**	**Res.**
Lang	testNullToEmptyIntObjectEmptyArray ArrayUtilsTest	6	low	low
Lang	testNonEquivalentAnnotationsOfSameType AnnotationUtilsTest	8	-	low
Lang	testCompare ObjectUtilsTest	8	low	low
Lang	betweenExclusive_returns$_false$ A_is_1.B_is_1.C_is_0	8	-	low
Lang	testOrdinalIndexOf StringUtilsEqualsIndexOfTest	24	-	low
Lang	testGeneratedAnnotationEquivalentToRealAnnotation AnnotationUtilsTest	16	-	high
Lang	testLocaleLookupList_LocaleLocale LocaleUtilsTest	25	-	low
Lang	testWriteNamedFieldForceAccess FieldUtilsTest	19	-	low
Math	testGetLInfDistanceSameType RealVectorAbstractTest	7	-	low
Math	dimensionCheck AdamsMoultonIntegratorTest	13	-	low
Math	testAdd1 WeightedObservedPointsTest	9	-	low
Math	testIsNaN Decimal64Test	8	-	low
Math	testGLSEfficiency GLSMultipleLinearRegressionTest	37	-	high
Math	testStandardTransformFunction FastFourierTransformerTest	21	-	high
Math	testStartSimplexInsideRange MultivariateFunctionMappingAdapterTest	23	-	high
Math	testThreeRedundantColumn MillerUpdatingRegressionTest	32	-	high
Mockito	no_op_when_no_mismatches StubbingArgMismatchesTest	11	-	low
Mockito	shouldArraysBeEqual EqualsTest	6	-	low
Mockito	test$_invoke$ MemberAccessorTest	8	-	low
Mockito	testReflectionHierarchyEquals EqualsBuilderTest	31	-	low
Mockito	should_delete_listener StubbingLookupListenerCallbackTest	30	-	low
Mockito	shouldAllowToExcludeStubsForVerification VerificationExcludingStubsTest	18	-	high

Table A.6: Second framework verification

Project	**Test & Class**	**FW**	**Dev**	**Res.**
Time	testMergePeriod_RP2 TestMutablePeriod_Updates	13	low	low
Time	test_getValue_long TestMillisDurationField	6	low	low
Time	test_wordBased_de_parseTwoFields TestPeriodFormat	12	low	low
Time	testAddYears TestMutablePeriod_Updates	10	low	low
Time	testPlus_RP TestLocalDate_Basics	10	low	low
Time	testForFields_weekBased$_D$ TestISODateTimeFormat_Fields	23	low	low
Time	test_set_RP_int_intarray_int TestPreciseDateTimeField	20	low	low
Time	testGetDurationMillis_Object1 TestStringConverter	17	low	low
Time	test16BasedLeapYear TestIslamicChronology	19	low	low
Time	testFactory_standardMinutesIn_RPeriod TestMinutes	16	low	low

Table A.7: Test categorization per project

Project	**Tests**	**Low level**	**% low level**	**High level**	**% high level**
Chart	2244	1708	76.11%	536	23.89%
Cli	235	221	94.04%	14	5.96%
Codec	842	800	95%	42	5%
Collections	1680	1463	87.08%	217	12.92%
Compress	1250	1085	86.8%	165	13.2%
Csv	343	319	93%	24	7%
Gson	1053	974	92.5%	79	7.5%
JacksonCore	729	464	63.65%	265	36.35%
JacksonDatabind	2652	1933	72.89%	719	27.11%
JacksonXml	305	292	95.74%	13	4.26%
Jsoup	862	442	51.28%	420	48.72%
JxPath	331	316	95.47%	15	4.53%
Lang	3320	3072	92.53%	248	7.47%
Math	3168	2451	77.37%	717	22.63%
Mockito	2215	1365	61.63%	850	38.37%
Time	4248	3079	72.48%	1169	27.52%

Table A.8: Triggered tests categorization per project

Project	Tests	Low level	% low level	High level	% high level
Chart	85	42	49.41%	43	50.59%
Cli	37	33	89.19%	4	10.81%
Codec	37	37	100%	0	0%
Collections	3	3	100%	0	0%
Compress	69	56	81.16%	13	18.84%
Csv	22	22	95.65%	1	4.35%
Gson	33	32	96.97%	1	3.03%
JacksonCore	48	19	39.58%	29	60.41%
JacksonDatabind	116	76	65.52%	40	34.48%
JacksonXml	12	12	100%	0	0%
Jsoup	131	71	54.2%	60	45.8%
JxPath	29	27	93.1%	2	6.9%
Lang	92	72	78.26%	20	21.74%
Math	101	47	46.53%	54	53.47%
Mockito	109	56	51.38%	53	48.62%
Time	73	51	69.9%	22	30.1%

Table A.9: Setup usage per project

Project	Total test classes	Classes with setup	% of classes with setup
Chart	341	24	7%
Cli	25	10	40%
Codec	66	5	7.57%
Collections	204	32	15.7%
Compress	147	14	9.52%
Csv	20	4	20%
Gson	95	41	43.15%
JacksonCore	141	0	0%
JacksonDatabind	557	2	0.36%
JacksonXml	113	6	5.31%
Jsoup	43	5	11.628%
JxPath	41	16	39.02%
Lang	198	30	15.15%
Math	358	27	7.54%
Mockito	464	73	15.73%
Time	127	95	74.8%

Table A.10: Big setup usage per project

Project	Classes with setup	Big setup	% of classes with big setup
Chart	24	4	16.66%
Cli	10	7	70%
Codec	5	2	40%
Collections	32	17	53.12%
Compress	14	9	64.28%
Csv	4	2	50%
Gson	41	1	2.43%
JacksonCore	0	0	0%
JacksonDatabind	2	1	50%
JacksonXml	6	2	33.33%
Jsoup	5	1	20%
JxPath	16	12	75%
Lang	30	8	26.66%
Math	27	10	37.03%
Mockito	73	20	27.4%
Time	95	50	52.63%