MASTERTHESIS

An Empirical Study of Testability in JavaScript Projects

by

Finn Carstensen

MIN-Fakultät

Fachbereich Informatik

Studiengang: Informatik

Matrikelnummer: 7065446

Abgabedatum: 03.09.2023

Erstgutachter: Prof. Dr. Walid Maalej

Zweitgutachter: Dr.-Ing. André van Hoorn

Betreuer: Pavel Reich

Abstract

Testability is a quality attribute of software that enables an efficient testing process where faults are likely to be found. Testability has been a subject of scientific research for decades, but there are gaps to be filled. Most research in this field focuses on Java, which is not the only relevant language in today's software development. This thesis focuses on JavaScript as a versatile language that is frequently used in many frontend and backend applications.

The goal of this work is to answer two main research questions. The first question is the following: What is the level of testability in JavaScript open-source projects and what are the characteristics of testable JavaScript? The question complements existing research for Java to analyze the impact of JavaScript's particularities on how testability is reflected in the source code. Additionally, observations about the testability level across the analyzed projects are made. An empirical study on a newly created dataset of 384 public open-source JavaScript and TypeScript repositories is done to find source code metrics that indicate the level of testability. The creation process was designed for reusability in future studies. 39% of the repositories contained at least one test case. Testability metrics were selected by performing a correlation analysis between pairs of test and source code metrics. A scoring system built on the resulting set of 34 metrics is proposed that can be applied by a developer to assess the testability of source code. The findings are that TypeScript appears to be more testable across open-source projects and that the level of testability seems to depend on the application area. Also, projects that contain test cases appear to have higher testability on average.

The second research question is: What refactorings are done in JavaScript open-source projects to enhance testability? Complementing a recent study on Java projects, the goal was to find out how developers improve testability in JavaScript open-source projects and how this information can be used to guide developers in refactoring their code for better testability. In a semi-automated process, refactorings were searched in a set of 802 commits that modify related test and source code files. Every commit was manually reviewed to identify refactorings for testability. In total, 42 testability refactorings have been found. Derived from them, ten refactoring procedures are proposed to improve testability in certain scenarios.

Acknowledgements

I would like to mention some people who supported me in creating this work. First I would like to thank my supervisor, Pavel Reich, who shared his experience with me and also reviewed some data to improve the meaningfulness of the results I found. I am also thankful for the feedback and guidance in finding a suitable topic for this thesis from Prof. Dr. Walid Maalej. The remarks of my friend Michael Raykhlin helped me a lot in improving the readability of this work. Last but not least, I would like to thank Mosabbir Khan Shiblu, who gave me access to the refactoring mining tool he developed for his master's thesis.

Contents

Abbreviations **1**

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Research Questions . . . 4
1.3 Structure Overview . . . 5

2 Fundamentals **6**
2.1 Testing of Software . . . 6
2.2 Testing in Open-Source Software . . . 11
2.3 Language Features of JavaScript . . . 12
2.4 Testing in JavaScript . . . 14
2.5 Testability of Software . . . 15
2.6 Metrics for Testability Measurement . . . 17
2.7 JavaScript on GitHub . . . 21
2.8 Refactorings for Testability . . . 21

3 Approach **24**
3.1 Sampling Strategy . . . 24
3.2 Testability Analysis . . . 27
3.3 Selection of Testability Metrics . . . 28
3.4 Finding Refactorings . . . 33

4 Analysis **36**
4.1 Dataset . . . 36
4.1.1 Meta Data . . . 37
4.1.2 Manual Classification . . . 39
4.2 Testability in JavaScript-Projects . . . 41
4.2.1 Collection of Metrics . . . 42
4.2.2 Correlation Analysis . . . 44
4.2.3 Testability Level Analysis . . . 46
4.2.4 Comparison with Java . . . 51
4.3 Refactorings for Testability . . . 52
4.3.1 Finding Commits . . . 52
4.3.2 Mining Refactorings . . . 52

4.3.3 Commit Analysis 55
4.3.4 Testability Refactoring Procedures 58
4.3.5 Comparison with Java Patterns 66

5 Conclusion **68**
5.1 Threats to Validity 69
5.2 Future Work 70

References **71**

Appendix **81**

Abbreviations

AC Afferent Coupling. 31, 32
API Application Programming Interface. 13, 14, 69
AST Abstract Syntax Tree. 9, 24, 42, 43

BCOV Branch Coverage. 32, 33

CC Cyclomatic Complexity. 30–32

DIT Depth of Inheritance Tree. 28
DOM Document Object Model. 14, 39, 65

EC Efferent Coupling. 31, 32
ECMA European Computer Manufacturer's Assiciation. 13
ES ECMAScript. 13, 14, 56

FCOV Function Coverage. 32, 33

GUI Graphical User Interace. 9, 10, 15

HB Halstead Bugs. 31, 32
HD Halstead Difficulty. 31, 32
HE Halstead Effort. 31, 32
HL Halstead Length. 31, 32
HT Halstead Time. 31, 32
HTML HyperText Markup Language. 65
HVOC Halstead Vocabulary. 31, 32
HVOL Halstead Volume. 31, 32

JS JavaScript. 39
JSON JavaScript Object Notation. 24, 37
JSTS JavaScript Testability Score. 69
JSTT JavaScript Testability Tool. 5, 24, 27, 36, 42, 44, 48, 53, 55, 69

KISS Keep It Simple, Stupid. 16

LCOM Lack of Cohesion in Methods. 28
LCOV Line Coverage. 32, 33
LOC Lines Of Code. 12, 25, 30–32
LOCC Lines Of Code Comments. 30, 32

LOCL Logical Lines Of Code. 30, 32
LOCR Test Lines Of Code to Lines Of Code Ratio. 33

MC Method Calls. 31, 32
MI Maintainability Index. 31, 32
MVC Model View Controller. 63
MVVM Model View ViewModel. 63

NBI Number of Bytecode Instructions. 28
ND Nesting Depth. 31, 32
NOC Number of Classes. 30
NOF Number Of Functions. 30, 32
NOM Number of Methods. 30
NOP Number Of Parameters. 31, 32
NOTC Number Of Test Cases. 32, 33
NPM Node Package Manager. 24
NPM Number of Public Methods. 28
NPRIM Number of Private Methods. 28
NTC Number of Test Cases. 12, 19

PAP Percent Public And Protected. 28

RBT Risk-Based Testing. 7
REST Representational State Transfer. 13, 24, 36, 53, 69

SBST Search-Based Software Testing. 18
SCOV Statement Coverage. 32, 33
SDLC Software Development Lifecycle. 6–8
SOLID Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion. 16
SUT System Under Test. 6, 9–11

TDD Test-Driven Development. 3, 11
TLOC Test Lines Of Code. 19
TRs Testability Refactorings. 58, 59
TS TypeScript. 39

UI User Interace. 9, 15, 27, 55, 58, 65, 67

VCS Version Control System. 21

XSS Cross-Site Scripting. 14

1 Introduction

1.1 Motivation

Probably every developer should know that testing is a crucial part of software development. [6, 32, 64, 67] The task of testing is time-consuming and comes with different challenges and difficulties. Plenty of research was done in the past decades dealing with topics like types of tests, principles, automation, test generation, testing in practice and much more. [36] Depending on the requirements for software, testing can consist of unit testing, performance testing, security testing and integration testing, to name the most common. [44, p. 10] For certain types of testing, the architecture and programming style of the underlying software are of great importance. [25, 33, 19] Guidelines for writing testable software and principles like Test-Driven Development (TDD) aim for a minimum effort in testing source code. [63] If source code is written without testability in mind, it may require changes when tests are added later. Even though academia paid attention to the testability of software, there are some gaps to be filled.
Testability is an important software characteristic. Testing already consumes up to 50 percent of the development effort [55, p. 1][64, p. 9], so it should be avoided to unnecessarily increase it by not thinking about testability when writing code. It will take longer to write tests if the source code is hard to understand. More complex tests will be needed if the source code lacks cohesion or has high coupling. Testability issues can range from architecture to implementation level. [19, p. 437] Building testable software will save resources and speed up the development process. [13, p. 87] Another problem is that tests will reveal fewer faults when the testability is low. [2, 35] As an example, think about a complex method with hundreds of possible paths depending on the input. It is unlikely that test cases will be written for every path, so problematic inputs may remain uncovered. If the method is split into parts with only a few paths each, it is easier to cover them in individual tests for each part. If software has low observability, faults that appear during the computation may remain undetected if the output is still correct.
When reading through the literature, the first thing to be noticed is that some programming languages get less attention than others. Much research deals with software written in Java, which stands out when reading systematic literature reviews like the work from Garousi et al. [35] There are examples of literature where authors explicitly recommended that researchers should examine other languages than Java too, like the work of Kaur and Singh regarding refactoring activities on GitHub. [52, p. 40] Research for JavaScript regarding testability seems to be rare. Even though Java and JavaScript share part of their names, they were designed for different purposes and use different concepts

and syntax. According to GitHub's yearly Octoverse reports [40], JavaScript is still the most frequently used language in GitHub repositories since 2014. It is versatile, it is used in most of today's websites to enable functionality, it can be used to create scripts and automation and even to set up web servers. On the other hand, some language features make it prone to errors, which confirms the need for proper testing. Due to the undebatable importance of JavaScript it should gain similar attention in software development research. TypeScript also made its way up to the fourth place in the ranking of popular languages in the Octoverse report, since it entered the top ten in 2017. Because it is a superset of JavaScript, it may be interesting to include it in research for JavaScript.

1.2 Research Questions

This thesis shall answer two main research questions to fill some of the mentioned gaps. For simplification, in the following the subject of investigation is referred to as JavaScript projects. But to be exact, this includes extensions and modifications of the JavaScript language like TypeScript. Also, the term does not refer to all projects that exist in the world. The limitations will be explained in chapter 3 in more detail.
The first question emerges because it is investigated for languages like Java but not for JavaScript. The question is: **What is the level of testability in JavaScript open-source projects and what are the characteristics of testable JavaScript?** Like maintainability, testability is nothing that can easily be measured. It is often described as consisting of properties like controllability, observability and simplicity. [30, 35, 32] There is no simple metric or aggregate of metrics that allows exactly measuring how testable software is, but it may be possible to find metrics that can indicate if software fulfills a certain aspect of well-testable software. That could either be achieved theoretically by finding metrics that go together with concepts of testable software or practically by analyzing existing code. Other studies already found some metrics that seem to be related to testability for Java code, for JavaScript the question remains unanswered. Due to the differences between the two languages, it is expected that results for Java do not necessarily apply to JavaScript too. It is an interesting question if there are differences between Java and JavaScript in this regard.
As in any field of software development, people strive to make processes as efficient as possible. Repetitive tasks should be avoided and swarm intelligence should be utilized to avoid common mistakes and save time. In the context of testability, Reich and Maalej already started examining patterns of refactorings. [71] They analyzed pull requests from GitHub to find refactorings that were done to improve testability. They focused on Java code and found ten composite refactorings. This work serves as an inspiration for the second research question in this thesis and allows a comparison of the results. The approach to finding those refactorings will be different though, reasoned by the use of a different dataset and the disadvantages of the other approach. In summary, the second

research question is: **What refactorings are done in JavaScript open-source projects to enhance testability?** If possible, patterns will be extracted from the findings to guide developers in similar situations and for tool support.

1.3 Structure Overview

Chapter 2 will give an overview of the research and concepts that form the basis of this work. First, aspects of testing that are important to understand testability are explained, starting with a general perspective and ending in the context of the JavaScript language. Next, the term testability is described in more detail and methods to assure and improve testability are presented. This includes existing approaches to measuring testability, with a focus on computable software metrics. At the end of the chapter, existing research on improving testability through refactorings is summarized.
Chapter 3 describes the approach that is followed in this thesis to achieve the two research goals. This starts with the generation of a representative dataset for the selected sampling frame. This work will use a newly assembled dataset to reflect the current state of open-source JavaScript projects on GitHub. With this dataset, it is described how testability is analyzed through the collection of software metrics with subsequent correlation analysis.
Chapter 4 covers the main work of this thesis. The process of obtaining the information of interest is explained step-by-step. For a structured and clear workflow and a comprehensive presentation of collected data, a tool called JavaScript Testability Tool (JSTT) is developed. The data is evaluated and the findings are explained.
The last chapter summarizes the work by critically looking at the results and discussing limitations. The latter include problems that have occurred and may have a negative impact on the results. It is also discussed how the findings can be used for future work.

2 Fundamentals

To get deeper into the subjects of this thesis, some general knowledge about concepts and associated literature is necessary. First, it is important to know what testing software means and why it is an essential part of software development. It is shortly introduced, which different techniques, methods and types of testing exist. It is described what challenges and particularities emerge in the context of JavaScript testing. Second, an overview of the testability aspect of software is given. That includes how it is defined throughout the literature and how it can be measured. Then, it is briefly explained how JavaScript is represented on GitHub. The chapter ends with an overview of existing work on refactorings for testability.

2.1 Testing of Software

Software testing is the process of evaluating if the System Under Test (SUT) meets its specified requirements. [46, p. 177] It verifies the quality of software by systematically exercising it. [55] Testing can find errors in a program, but can never prove the absence of all errors. [66, p. 52] That is why it always makes sense to add missing test cases, even in later phases of the Software Development Lifecycle (SDLC). It follows, that improving the testability of software is useful at any point in time too. In safety-critical applications, formal and mathematical methods that aim to prove the correctness derived from the specifications are used. [66, p. 54]
The short-term goal of testing is to find faults in the system that lead to unexpected behavior, which can be fixed afterward. The long-term goal is to ensure that the system meets its requirements, even after code changes. [66, p. 20] This section will address fundamental aspects of software testing to provide a general overview. The subject of this thesis is testability, which is strongly connected to testing, but testing covers many more topics that are not directly affected by testability. For this reason, this section will not cover the planning or execution of every facet in detail but give a high-level overview of testing.
Software testing builds upon the specification of the software, which is the obligatory document for stakeholders defining all the settled requirements. The specification includes functional requirements which describe the expected functionality and non-functional requirements which address concerns regarding aspects like performance, availability, security and maintainability. [66, p. 48] The requirements have a very important role in software testing since test developers need to know what to test for. The quality of the specification and documentation artifacts resulting from the planning phase

of a project has a great impact on the quality of the test suite. [26, pp. 21–22] Tests are not an extension or substitute for specification though, because they lack the needed abstraction and are not accessible to everyone, especially stakeholders not involved in the coding process. [60] Expected and unexpected behavior needs to be well-documented so that the tests are likely to reveal faults. If so, tests could also be automatically generated from the specification. Motwani and Brun presented an approach to extract test oracles from natural language specification. Test oracles are the data that is used in test cases, e.g., combinations of inputs and outputs. Obtaining those oracles can be hard, e.g., for complex algorithms. [62]
Software testing and development are depending on each other. [26, p. 16] Writing testable code is a precondition for executing tests and writing tests is necessary to ensure a unit is working correctly before it is used further. In a case study, Ghafari et al. analyzed the relation of bugs to test coverage of software by looking into bugs described in issue trackers. They found that only very few components affected by bugs were tested. [38, p. 3] It confirms that testing eliminates defects. However, their findings may not apply in general since it was only a case study.
While testing is very important for the quality of the underlying software, it can be a costly process that needs to be well-managed and planned. Testing usually needs about half the time of the development effort. [55, p. 1] Often a tradeoff between the testing effort and a feasible amount of resources for testing needs to be made. [46] Resources are often limited and deadlines can negatively impact testing when the production code to be tested is not handed to the test developers on time. [66, p. 61]
The minimum effort for testing highly depends on the criticality of the software, e.g. software in a hospital that is used to evaluate the vitals of a patient is more important to be well-tested than a blog on the internet. Before a software development project even starts, the risks are evaluated. Risk assessment is an important process that can essentially influence the business success of software. [28] It is questioned if and how much testing can reduce the identified risks. In monetary terms, the cost of mitigating risks should not exceed one-fifth of the potential loss. [26, p. 10] Testing 100 percent of the software is unrealistic, so the tests need to reduce the risks as much as possible within the given resources. [26, pp. 9–11][66, p. 71] Therefore, a prioritization of tests needs to be done to examine the point at which the existing tests are sufficient. The reduction of the highest risks should be the highest priority in testing. This approach is called Risk-Based Testing (RBT). [28, 70] RBT can be kept in mind to understand why some parts of software are better tested than others. Generally, the business risk of developing software can be reduced by reducing the number of defects in the software. [26, pp. 17–18]
A popular rule of thumb is that the cost to correct an issue in the code increases significantly the later it is done within the SDLC. However, a study by Menzies et al. from 2017 disproves this rule in an empirical study, showing that the effort to remove defects

was not significantly greater in later phases of the SDLC of recent software projects, at least in the used sample. [58]
There are ways to measure the effectiveness of test suites. A common technique is to compute the code coverage of tests. Statement coverage simply checks which and how many lines of the source code were executed while running the test cases. Branch coverage reflects which and how many logical branches of the code were executed, branches emerge whenever the next statement for execution is dependent on a condition. Path coverage takes this even further and reflects which and how many individual paths of statements were executed. A path can contain multiple decision points and full path coverage can therefore reveal more defects than just full branch coverage. [26, pp. 107–110]
Testing is an activity that should be performed throughout the whole SDLC. It can be separated into four different phases: Unit testing, integration testing, system testing and acceptance testing. Unit testing is the testing of smaller units of software in isolation, to guarantee each delivers the expected behavior. Integration testing takes place when modules are put together to test if a larger task works as expected. System testing is the testing of the software as a whole from each possible perspective and is often based on the specification of the system. [46, 55, 66] Acceptance testing ensures that the user is satisfied with the software product and accepts to use it. [66, p. 51]
There are three major testing techniques, black box testing, white box testing and grey box testing. [48, pp. 31–32] White box testing can apply to all of the mentioned testing phases and means that the test developer has full insight into the production code. It tests the actual functionality of the code while challenging the different execution paths. [55, p. 3] Black box testing does not use implementation details, but can still be applied to all testing phases. It tests the software by checking if the initially defined requirements are met. [55, pp. 2–3] The software can still be executed but without insight into the actual implementation. [26, p. 68] Grey box testing tries to use the advantages of the two other techniques. Tests are written with some knowledge about internal structures but treat the unit under test as a black box.
Another distinction in testing can be made between positive and negative testing. Positive testing is verifying that valid interactions or inputs lead to the expected results. Negative testing is verifying that incorrect but possible interactions or inputs do not break the software and lead to unwanted effects. Positive testing is often higher prioritized in software development since it is easier and less time-consuming. [26, p. 18] Nevertheless, users often tend to interact with a system in unexpected ways. Since they may not completely understand the system as well as a developer does, they will use the system differently. The discrepancy between the mind of a user and a developer makes it hard for the latter to find all the negative test case scenarios users may produce.
When speaking of software testing, often execution testing is meant. Execution (also dynamic) testing is running the software to get test results. There also exists another

method that does not require executing the software, which is static testing. Static testing can be done on a vast amount of artifacts, like requirements, specification documents or the source code. The testing of documents can remove many problems even before the source code is written. [26, p. 18] Static testing of the source code can improve code quality without any information on the runtime behavior. [55, p. 2] It often uses the Abstract Syntax Tree (AST) representation of the code to find problems. For example, many IDEs or editors use linter tools to give the developer instant feedback on code smells or syntax mistakes.
Testing is not all done manually, reducing cost and effort. For example, regression testing automates the process of running tests after code changes, allowing for instant feedback if something breaks and making sure no new bugs are introduced into the software. [26, p. 106] Mutation testing is the testing of the test suite itself, it generates versions of the source code with small faulty changes and checks if the test suite runs through. [61, p. 429] The research interest in test automation like this has been high over the last decades.
So far, requirements used for testing were not considered in detail. Requirements can address different aspects of software. Requirements can address functional correctness, performance or security. Each aspect needs separate tests. Functional tests mostly check the outputs against inputs. An input could be a value passed to a function or an interaction with a User Interace (UI). The output does not necessarily need to be a return value but can also be a state change. A Graphical User Interace (GUI) is a challenging subject of software testing because GUIs are often changing frequently. [69, p. 1] GUI tests differ from other functional tests and need to be considered separately.
Performance testing verifies if the SUT meets the constraints for response time and throughput. Such constraints are important because users will be frustrated if they have to wait for a long time to get a response from the system, for some systems this will lead to economic damage. Performance testing should be done after functional testing when the correctness of the SUT is on a sufficient level. It is considered more difficult than functional testing since it only gives useful results if done correctly. [26, p. 69] E.g., the response time of a function can not be measured representatively if it is only executed once. The response time may vary depending on the underlying operating system, the amount of other parallel tasks or background processes like garbage collection. The environment of performance testing needs to mimic the environment where the software will run in production as closely as possible. In performance testing, peak loads of the system are of special interest. The software may have timeframes where it is used more extensively and is expected to still satisfy the user's needs. [26, p. 129] E.g., an online store can recognize a heavy load on Black Friday sales with way more concurrent sessions than usual. If the system breaks and customers are not able to checkout, this will lead to economic damage for the vendor. If performance constraints are not fulfilled, the code may be inefficient and needs performance optimizations or the underlying hardware

needs to be upgraded.
Security testing is often also crucial and should guarantee that only authorized interactions with a system are allowed, that data is only accessible by authorized actors and that information transfer is encrypted safely. One technique in security testing is taint analysis, which examines the flow of data in software to find vulnerabilities that could lead to Cross-Site Scripting (XSS) or the exposure of sensitive data such as passwords. [7, p. 9] However, security testing will not be a subject of this thesis, since it is often done with external software that statically scans artifacts for vulnerabilities. Thus it will not be reflected in available artifacts of open-source software repositories. Testability is not a criterion for those scanners by design because they can analyze any code and check for known vulnerabilities. A recent study by Brito et al. found that most automatic security vulnerability detection tools for JavaScript suffer from very low precision, indicating that this field needs improvements. [17] Other tools utilize test suites, but it is unlikely to create meaningful results when most of the security testing is missed.
When talking about testing, a technique following another approach needs to be mentioned, which is using assertions. They are constraints on the behavior of software that can find faults in programs during runtime. [72, p. 19] They are a mechanism to improve the observability of software. The drawback is that they can negatively impact the performance of a system since they are evaluated at runtime. That is why Voas and Miller proposed to only use assertions where other testing techniques are limited. Additionally, assertions may be removed from the code after it has been tested sufficiently. [82, pp. 152–152]
Testing can be done manually or automatically. Manual testing is a resource-intensive task but can be useful in certain scenarios. It is common practice to automate as much as possible in the testing process. [51, p. 1] Frameworks for writing, running and evaluating test cases exist for the majority of programming languages. Over the years, many tools for test automation have evolved. They follow different approaches and address different problems, so the selection of the right tools requires some knowledge. [81, p. 223] Like manual testing, a prerequisite for test automation is that the source code is testable. For poor code and architecture, automated testing will not be beneficial. For this reason, automation can also be used to test if the code is testable. [51, p. 3] For GUI testing, automation is particularly hard because the appearance or layout can change very often and devices can have different dimensions. That often leads to the necessity of changing the associated test code. Ensuring a high level of testability is particularly important for GUIs, because this way test automation can be more effective and the effort for this resource-intensive task can be minimized. Polepalle et al. found that requirements for high testability in general also apply for GUIs. [69]
Another important step in creating tests is the generation of test data. This process should not only rely on the manual test cases the developer came up with. They may test a certain aspect of the SUT thoroughly but might miss other important cases. So the man-

ual test cases should be complemented with automatically generated ones. [60, p. 100] Nowadays, there are plenty of tools that do much of the work in test generation, aiming for a high code coverage and using knowledge about common edge cases collected over decades. Algorithms for automatic test data generation have existed for a long time, e.g., evolutionary algorithms. [24] Ferguson and Korel differentiate between three types of data generators which are random, path-oriented and goal-oriented. The random technique generates random data. Path-oriented data generation analyzes a program and generates data that challenges specific paths. Goal-oriented data generation creates data that leads to the execution of specific statements, independent from the path. [29, p. 64] Also, random testing does a good job of finding faults that manually generated test cases did not get. [60, p. 100] Fuzz testing is an established approach, that uses random as well as unanticipated and invalid inputs to test the reliability of the SUT. [21, p. 210]
The level of testability influences the required resources for testing since a testable program does not need refactorings to allow the execution of new tests. A famous concept in software engineering is Test-Driven Development (TDD). It came up in the context of extreme programming and tried to shift more attention to testing. It corresponds to unit testing and requires the tests to be written before the production code. Each time a new functionality should be added, the tests need to be written first. Studies have shown that this process reduces faults, increases code quality and can even speed up development. [22] TDD kind of forces the developer to write testable code, so the expectations. Since the test already exists, the code must be testable by the underlying tests. Müller introduced assignment controllability as a metric to measure testability and finds a positive correlation to the use of TDD. [63] George and Williams conducted a case study with two groups of programmers, one using TDD and the other writing tests after writing the production code. The TDD group produced higher quality and better testable code but needed more time to achieve that. [37]
As already stated, there is a lot more to say about software testing, but considering every aspect would go beyond the scope of this thesis. The goal was to give an overview of what testing can consist of and how testability influences testing. This is necessary when tests in real-world projects should be analyzed, which is part of the work of this thesis.

2.2 Testing in Open-Source Software

The principles and aspects of testing described in the previous section are mostly tailored to classic software development, where a need for software exists in a company and an in-house team or an external software house is instructed to implement it. For open-source software, the whole development process is different from proprietary software. An open-source project may be started due to the personal needs of one or more people, or it may be initialized by a company with economic interest. Often, open-source projects aim to solve a common problem. Contributors usually do not get paid, so their

motivation emerges from their interest in being able to use the software. [14]
Open-source development is often community-driven and may lack concrete specification documents. [23, p. 136] There may be sparse and informal documentation about what the software does and how to use it, but it usually does not cover many details. The software is often evolving through feature requests or bug reports from the community. That is why those artifacts are an important part of the specification of the project. The issue is that it can be challenging to locate the desired information in this scattered knowledge base. Because of the differences in project planning and documentation, the testing process also differs. If no detailed specification exists, test cases can not be derived from it. That shifts the responsibility for testing to the individual contributors. There is most likely no person or team dedicated to testing. When browsing through open-source repositories, one can often find guidelines that instruct contributors to test their changes or features themselves. New features can easily be proposed by community members to be added to the production code. Contributions often get reviewed by the community or some main contributors, which serves as quality assurance, also regarding tests. In open-source projects, the tests are often part of the specification, when the reflected requirements are not written down elsewhere. In open-source development, the repository on a platform like GitHub is the main artifact that is accessible to everyone. Maintaining other artifacts like specification documents can be difficult due to the organization of the project. So keeping all the information here can also be beneficial in some ways.
An empirical study by Kochhar et al. used a corpus of more than 20,000 open-source software projects from GitHub to analyze the prevalence of tests and correlations to the number of developers, the number of bugs and bug reportings and the used programming language. They found that bigger projects in terms of Lines Of Code (LOC) are more likely to have tests, but the Number of Test Cases (NTC) per LOC decreases with growing project size. 61.65% of projects did contain test cases. [53] Madeja et al. did a similar empirical study, focussing on the correlation between the prevalence of test cases and the identification of corresponding resources using the word ”test”. Using a dataset of 6.3 million GitHub projects with Java as their primary language, they found that 51.57% contained at least one test case. However, they noticed that an estimated 67% of projects with at most two test cases only contain example test cases, causing them to estimate that only 38.84% contain actual test cases. [57]

2.3 Language Features of JavaScript

JavaScript, often referred to as the language of the web, is an indispensable technology in today’s software infrastructure. It is the default for browsers, for which it was initially invented. Technologies like NodeJS[1] allow server-side execution of JavaScript

[1] `https://nodejs.org/`

and therefore the building of applications like REST APIs. There exists a standardized specification of the JavaScript language, namely ECMAScript or just ES.[2] Different institutions implemented their own versions of the standard, which differ from each other on the implementation level. Popular JavaScript engines are SpiderMonkey[3] and V8[4].
JavaScript is object-oriented but has some features that make it different from other languages. It is interpreted, meaning that code is evaluated at runtime and the execution depends on the source code rather than binary files. It is dynamically typed, meaning that the type of an object is determined at runtime. Type coercion allows for the runtime conversion of types, e.g. using a number as a string. [21, p. 59] There exists a famous extension of JavaScript that adds types to the languages, which is TypeScript. All objects in JavaScript are of the same structure, they hold a dynamic set of properties. Additionally, everything in JavaScript is an object, even functions. JavaScript is prototype-based, which is significantly different from statically typed, class-based languages like Java. [7, p. 19] Java or C++ use classes to describe objects comparable to blueprints that can be extended to implement inheritance. In JavaScript, prototypes are used as a mechanism that handles encapsulation, inheritance and polymorphism. [21, p. 20] In Java, encapsulation is implemented through the use of the keywords *private*, *public* and *protected*. JavaScript does not have a mechanism like this, but you can still hide properties from the public interface by using local variables inside objects or the #-prefix for class members. Properties defined on the object are instance properties, which can also be functions.
In terms of inheritance, there is a basic object prototype that every other object inherits. JavaScript uses prototype chaining for inheritance, meaning that a property is searched along the reference chain until it is found. JavaScript uses ad hoc polymorphism, meaning that the context of the call influences the result. Function calls can supply any number of arguments, independent of how many the function signature expects. [21, pp. 22–23] JavaScript applications do not use prototypes for code reuse as frequently as applications written in Java or C++ do. [86, p. 868] That influences the possibility of using metrics related to inheritance for the analysis of JavaScript code, which will be further described in the following chapters. Being a dynamically typed language also means that properties can be added or removed at runtime. Using ”duck typing”, determining if an operation can be done on an object only depends on the properties of an object [7, p. 4].
JavaScript's approach to executing code is the event loop, meaning that events are processed sequentially by listeners. The implementation of the run-to-completion concept takes care that functions are executed completely before other functions are invoked. [21, pp. 81–82] Only when waiting for asynchronous results, other executions can be

[2]https://www.ecma-international.org/
[3]https://spidermonkey.dev/
[4]https://v8.dev/

done meanwhile. JavaScript executes everything in a single thread, so no concurrency exists by default. In newer implementations of JavaScript, workers can be used to execute code in another thread, but they are limited to specific actions and do not share resources. [21, pp. 96–97]

2.4 Testing in JavaScript

An empirical study from 2017 by Fard and Mesbah found that 22% of JavaScript projects did not have any test code. There is a difference between frontend and backend code though. Server-side code is tested more often with only 3% of projects not having tests. The used dataset was selected from various sources and represented different domains. They also found that root causes for untested code are event-dependent or asynchronous callbacks and DOM-related code, because it may be harder to test. [27] Unfortunately, the sampling was not random and mostly included popular projects, thus it is not necessarily representative.

Testing JavaScript applications has some differences from testing applications written in other languages. Most of the research in testing focuses on object-oriented languages like Java and C++. JavaScript is object-oriented as well, but as described in the previous chapter it has its peculiarities. Due to its dynamic nature, static analysis of JavaScript is more difficult than in other languages. [7, 47] Especially interactions with the Document Object Model (DOM) are difficult to test, because DOM structures can get quite large and complex, and the code might rely on certain elements or structures. [7, p. 2] This challenge only exists in the frontend though, while many JavaScript applications represent server logic and do not face this issue.

JavaScript has some unique properties regarding performance testing. Since it is single-threaded, concurrency is not a concern in the testing of JavaScript applications. A problem is the variety of engines used to execute JavaScript. That particularly affects the performance, leading to different execution times of the same code on each engine. Performance optimizations that fit one engine can even worsen performance on another engine. Across the versions of one engine, there may be significant differences in performance too. [74] It is important to keep that in mind when testing the performance of JavaScript applications. The variety of JavaScript implementations also leads to the importance of another testing category, which is compatibility testing. ES explicitly allows adding language features, so there are APIs that are not consistently implemented across all engines. Compatibility testing ensures that the code can be executed across all targeted platforms. [59]

JavaScript has some unique security concerns. A good example of a feature that stands out here is the possibility to dynamically add and execute code with the *eval* function. Among other difficulties, this increases the likelihood of XSS vulnerabilities. [7, pp. 21–22]

For automatic test generation, two input spaces need to be considered, the event space and the value space. [73, p. 514] Since JavaScript's execution is event-based, different events or their order can influence the behavior. Hence, sequences of events need to be tested. The value space consists of values for the possible input types like strings, numbers or booleans. Saxena et al. proposed a tool that explores the value space using a concolic (concrete and symbolic) execution framework, consisting of a symbolic execution engine combined with a string constraint solver and addresses the event space through GUI exploration. [73] There are several other approaches to exploring the value space, often using similar concolic execution techniques. [7, p. 24]
This thesis includes TypeScript projects in the analysis since it is a superset of JavaScript. The static analysis of TypeScript is easier than it is for JavaScript because TypeScript delivers the type annotations that JavaScript is missing. For example, Park uses TypeScript definitions of JavaScript applications to allow more effective static analysis of the latter. [68]
Unit testing in JavaScript has no such standard as Java for example. Java has the fully integrated unit testing framework JUnit, whilst for JavaScript there exists a large variety of different tools. Like production code, test code can suffer from bad quality and code smells. Jorge et al. studied the presence of test smells in very popular JavaScript projects, finding that test smells can be found across all of them. [47]
For frontend JavaScript applications, there is the additional need to test the UI. There are established tools to assist this workflow like Selenium[5] or Cypress[6]. They allow the writing of test code for interactions with the GUI similar to writing common unit tests.

2.5 Testability of Software

The ISO/IEC 25010:2011 standard defines testability as the "degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met" [45] and sees it as part of maintainability. Efatmaneshnik and Ryan claim the consensus for testability to be "defined as the degree to which a component or system can be tested in isolation" [25, p. 1]. Freedman defines testability as a measure composed of the two attributes controllability and observability, which were originally used to assess the testability of hardware but are also applicable for software. [32] Since the goal of testing is to find faults, bugs and errors, testability can also be seen as "the tendency for software to reveal its faults during testing" [85, p. 1]. A testable program also means that the effort for automated testing is low and is a prerequisite for high-quality tests. [38] Improving testability usually has the goal of reducing the cost of testing and increasing the probability of finding faults in testing. [33, p. 1] Various authors present

[5] https://www.selenium.dev/
[6] https://www.cypress.io/

their definitions of testability while often focusing on specific dimensions related to their work, Garousi et al. list 33 different definitions found in a systematic literature review. [35, pp. 4–5] The different definitions show that testability is a broad term that can be analyzed in different dimensions.
In this thesis, testability is generally treated as the ease and the degree to which a system and its units can be tested to reveal their faults. If a system has a high level of testability, it means it is well-testable. The focus is on unit tests since testing a system as a whole is often not reflected in the available artifacts for open-source software, namely the source code. High testability means the effort to test a system is low and vice versa. That includes the generation of test data, test code and running the tests. If changes need to be made to the source code to test it properly, testability problems can be assumed. Testability is also generally known to be correlated to software quality, where higher testability leads to higher quality. [35, p. 2] Accordingly, the usage of tools that assist developers in writing higher-quality code can also improve testability. Many such tools for JavaScript like ESLint[7], JSHint[8] or SonarQube[9] are using static code analysis, some like DLint [41] also use dynamic code analysis. There are guidelines for writing testable code that go hand in hand with guidelines for writing good code in general. For example, Trostler mentions code readability, loose coupling and simplicity as important principles for writing testable code. [80] This reflects elementary coding principles like Keep It Simple, Stupid (KISS) or Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion (SOLID).
Gao and Shih define an analysis model for component testability, which is an aggregate of measures for five software attributes. The attributes are understandability, observability, controllability, traceability and test support capability. Understandability considers the documentation artifacts of a component, which are requirements specification, API specification and the user reference manual. [34] Since their approach acts on the component level, the measures analyze black box testability and do not consider the actual implementation. However, the documentation on the source code level (like code comments) can also contain helpful information for testing if the mentioned artifacts are insufficient, so they could be used for measuring testability as well. [2, p. 2] E.g., when there is no detailed interface documentation, code comments can show how to use and also how to test a software unit. In Gao and Shih's model, observability considers the observability of GUI events and interactions, functions, external interactions and message-based communication.
Controllability refers to the ability to control the environment, state of the component, execution, testing and functions. Traceability considers the format of program traces, the ability to insert or delete code for tracing, different trace types, trace collection and trace

[7] `https://eslint.org/`
[8] `https://jshint.com/`
[9] `https://www.sonarsource.com/products/sonarqube/`

storage. Traceability is further separated into five categories: Component operations, performance, states, events and errors. Test support capability refers to test generation, management, coverage analysis and execution. [34]
Testability is not a distinct aspect of software, it has similar goals to maintainability, for example. As testability is seen as an important aspect of software that can help reduce cost and improve efficiency and software quality throughout the phases of the SDLC [2, p. 2], measures of the degree of testability are necessary. Although testability is essential, it can be hard to evaluate, because it is composed of different aspects and might be subjective. [5, p. 55] As a precondition for testing, there must exist a well-defined set of requirements regarding the expected behavior of a software unit. It may not be easy to ensure the completeness of this set. Specification, especially regarding requirements, is an essential part of the testability of software. A testable software unit should allow for testing all of the requirements. But if there are no requirements specified, even well-testable source code cannot be properly tested.
Next, testability depends on suitable tooling. E.g., for units depending on external units or libraries, a mocking tool may be necessary. The skill and experience level of the test developer is an important factor too when it comes to understanding the unit under test. Additionally, the quality and variety of written test cases depend on the mind of the test developer, because when he just tries to show that the SUT is working correctly, he will come up with test cases that prove his belief. In contrast, a test developer that wants to find errors will write other test cases that challenge the program in all edge cases he can imagine. [66, p. 71]
Voas and Miller proposed sensitivity analysis as a technique to assess the testability of a system based on their definition of testability. It is composed of three elementary steps, which are observing the effect of a given input distribution on the original program, mutants of the program and the program with a modified data state. [82, 83, 84] Higher sensitivity means higher testability since high sensitivity means that software can reveal its faults. [33, p. 2]

2.6 Metrics for Testability Measurement

The first part of this empirical study consists of measuring the level of testability, therefore a foundation of testability metrics is needed. As already stated, there is no single metric that measures testability as a whole. Testability is - like maintainability - an attribute of software that is influenced by various factors throughout the software development process. Aside from the aspects of testability that are hard to measure, different studies proposed measures that can be computed reliably from the source code. A distinction into two types can be made, there are static metrics that can be computed by analyzing the source code, and there are dynamic metrics that can be computed by running the software. Most studies on testability metrics focussed on object-oriented

languages, leading to many metrics reflecting the relations between classes. [35, 2, 10, 11, 13, 18, 49, 50, 78] No recent studies focussing on metrics for assessing the testability in JavaScript, which is the subject of this thesis, could be found. Since JavaScript is not class-based, some metrics are not applicable in their original form. Nonetheless, some of the proposed testability metrics related to classes could be used for JavaScript, if they were adapted to reflect the individual properties of the language.

A literature review was done to collect metrics related to testability. Some empirical studies tried to find applicable metrics by calculating the correlations between test and source code metrics. Since there needs to be a basis of metrics to find correlations, often assumptions were made. There is quite a lot of literature on testability metrics in general. In the following, only studies that proposed metrics useful for the subject of this thesis are mentioned. Proposed metrics from those studies, that were found not to be useful, will be omitted as well. Literature that proposed metrics is briefly introduced.

Alenezi collected static metrics for test effectiveness, which is correlated to testability. [2]

Binder proposed metrics mostly for measuring the testability of classes in the object-oriented paradigm. [13]

Filho et al. recently found a set of metrics either from a black or a white box perspective. [30]

Bruntik and van Deursen used object-oriented metrics for predicting class testability. [18]

Trostler mentions some metrics connected to code complexity in the context of writing testable JavaScript. He describes low complexity as a main criterion for writing testable code. He sees a difference in the complexity of algorithms reflected in the code and the complexity of the code itself. It is likely necessary to have complex algorithms, but the surrounding code does not need to be complex as well. He also sees linting tools like ESLint as a provider of a good measure for testable code, since readability is connected to testability. [80]

Badri et al. studied the relationship between the testability and cohesion metrics to predict the testability of classes. [9]

Bajeh et al. empirically studied the suitability of object-oriented metrics as testability indicators. [11]

Arcuri and Geleotti used ”Testability Transformations” to improve Search-Based Software Testing (SBST) and verified their findings by using code coverage as a metric. [8]

Kasisopha et al. developed a machine-learning model to predict the testability level of classes using a selection of metrics as input variables. [50]

Nasrabadi and Parsa built a classifier model using a large set of software metrics belonging to the size, complexity, cohesion, coupling, visibility and inheritance of object-oriented software to predict the code coverage of automatically generated tests by pop-

ular frameworks. The predicted coverage was used as an indicator of testability. [65] Terragni et al. studied the relationship between test effort and object-oriented software metrics respecting test quality, which is quantified through code coverage and mutation score. [79]
Table 2.1 shows all potentially relevant metrics collected from the mentioned work. The first column shows the abbreviation. The second column shows the meaning. The third column shows the literature that used the metric and the last column connects the metric to the corresponding attribute of software. In the literature, the most frequently mentioned attributes that shall indicate the level of testability are controllability and observability, additional ones are simplicity and comprehensibility. [30, 35, 32] In general, most of the metrics used measure an aspect of the complexity of code. The metrics will be further evaluated in the next chapter to generate a final set of testability metrics, that can be used in the context of this thesis.
Especially two metrics, TLOC and NTC, were often selected as the indicators of testability to further find correlations with other metrics. [78, 18, 54, 30] The use of these metrics is reasonable, even if it does not necessarily need to reflect testability. Ideally, the test code to source code ratio should be one-to-one [2, p. 2], so a big test suite (in relation to the size of the corresponding source code) can indicate high testability. Nevertheless, a counterargument against the exclusive use of these two metrics can be found easily. If the test code is small, the software could just lack test cases or have poor test quality, even if the source code is well-testable. [79, p. 241] Therefore, other metrics need to be considered as well.

Table 2.1: Testability metrics from literature

Metric	**Description**	**Source**	**Attributes**
LOC	Lines Of Code	[2, 30, 65, 79, 80]	Size, Complexity
NBI	Number of Bytecode Instructions	[2, 79]	Size
LOCCOM	Lines Of Code Comments	[2, 79]	Understandability
TLOC	Test Lines Of Code	[18, 54, 78, 79]	Understandability, Testability
NTC	Number of Test Cases	[18, 30, 50, 78, 79]	Testability
TCOV	Test Code Coverage	[8, 79]	Testability
TBCOV	Test Branch Coverage	[79]	Testability
TMC	Number of Method Calls in Test Class	[79]	Testability
TWMC	Weighted Method Complexity in Test Class	[79]	Testability
TAMC	Average Method Complexity in Test Class	[79]	Testability

Continued on next page

Table 2.1 – *Continued from previous page*

Metric	Description	Source	Attributes
DEP	Dependencies of Tests	[30]	Complexity
NS	Number of Steps (in test case)	[30]	Complexity
DPT	Depth Test	[30]	Complexity
NTD	Number of Test Data (for each test case)	[30]	Complexity
Tassert	Number of Assertions in class under test	[9, 54, 79]	Complexity
TNOO	Number Of Operations in Tests	[54]	Complexity
CC	Cyclomatic Complexity	[30, 65, 78, 80]	Complexity
AVCC	Average Cyclomatic Complexity	[11]	Complexity
MAXCC	Maximum Cyclomatic Complexity	[11]	Complexity
TCC	Total Cyclomatic Complexity	[11]	Complexity
NMC	Number of Method Calls	[2, 79]	Complexity
LCOM	Lack of Cohesion in Methods	[9, 13, 30, 65, 79]	Complexity
HVOL	Halstead Volume	[11]	Complexity
LOCC	Lines Of Code per Class	[50]	Size
RFC	Response For a Class	[2, 13, 50, 79]	Complexity
TRFC	Test Response For a Class	[54]	Complexity
WMC	Weighted Methods per Class	[2, 13, 18, 30, 50, 79]	Complexity
AMC	Average Method Complexity	[79]	Complexity
TWMPC	Test Weighted Methods Per Class	[54]	Complexity
EC	Efferent Coupling (Import Coupling)	[11, 50, 78, 79]	Complexity
AC	Afferent Coupling (Export Coupling)	[11, 30, 65, 78, 79]	Complexity
CBO	Coupling Between Objects	[13, 50, 79]	Complexity
DCBO	Dynamic Coupling Between Objects	[78]	Complexity
NOC	Number Of Classes	[78, 30, 65]	Size, Complexity
NOM	Number Of Methods	[13, 30, 78, 11, 65]	Complexity, Encapsulation
PAP	Percent Public And Protected	[13]	Encapsulation
PAD	Public Access to Data members	[13]	Encapsulation
DIT	Depth of Inheritance Tree	[11, 30, 65, 79]	Complexity
NOA	Number Of Attributes	[30, 65]	Controllability
NOP	Number Of Parameters	[30, 65]	Controllability

Continued on next page

Table 2.1 – *Continued from previous page*

Metric	Description	Source	Attributes
NP	Number Of Preconditions	[30]	Controllability
NOST	Number Of Statements	[65]	Size
NPM	Number of Public Methods	[79]	Encapsulation
NPRIM	Number of Private Methods	[79]	Encapsulation

2.7 JavaScript on GitHub

GitHub is one of the most popular platforms for hosting code and other artifacts benefitting from a powerful Version Control System (VCS). As of today, a lot of open-source software hosts their code on GitHub, enabling developers from all over the world to easily participate in its evolution. While the open-source concept has a positive influence on many aspects of software development, it is also beneficial for studying facets of software. According to their website, GitHub hosts more than 300 million repositories [39] written in various programming languages. For every repository, a main language is predicted by GitHub itself using a third-party service. While that may lead to misclassifications in some cases, repositories are mostly labeled with the language that is mostly used. For JavaScript and TypeScript, GitHub holds a total of about 27 million repositories. How that information is retrieved is explained in section 3.1.
With its mature API, it is possible to automatically retrieve almost any kind of repository data, with some limitations due to usage constraints like rate limits. GitHub gives the user many features at hand, which can enrich software artifacts with useful information. For example, issues and pull requests can be labeled with some keywords to give a quick insight into their purposes. Nevertheless, many features are optional and are not heavily used in many projects. Even comments for, e.g., commits are not mandatory and do not have a standardized format. It can be any informal text that the developer thought about while publishing his work.
Smaller projects may tend to make less use of advanced features like labels or pull requests. In bigger projects, when more people need to work together and understand what others did, those features make life easier. For studies and analyses, it must be kept in mind, that consistent use of advanced features and documentative artifacts cannot be assumed.

2.8 Refactorings for Testability

Back in 2011, Harman introduced the term *testability refactoring* as part of testability transformation to improve the software's ability to support test data generation. It is an interesting approach because the goal is not only to make code more testable for a devel-

oper but also for automatic test generation. He describes ways to find suitable refactorings and advises future researchers to go deeper into the field of testability refactorings. [43]
Already over 20 years ago, lists of refactoring types like the one proposed by Fowler [31] came up to summarize and explain common refactoring practices. Testability refactorings are a subset of these and only certain types are likely to be able to improve testability. When studying the motivations behind refactorings in GitHub repositories, Silva et al. found that improving testability is indeed an existing motivation for the *extract method* refactoring, even though the number of occurrences is low compared to others. [75, p. 863] This finding indicates to watch out especially for this type of refactoring when searching for testability refactorings.
Cinnéide et al. studied an automated refactoring approach to improve software testability. They used a tool to automatically refactor software regarding design issues and asked software engineers to write test cases before and after refactoring. Their study design might have some problems because they only used one self-constructed example application for which the developers should write tests. In contrast to their expectations, developers found the refactored application not easier to test. [19] Since not all refactorings are refactorings for testability, it is not clear if the ones they did can be considered testability refactorings. Otherwise, the developers should have had more problems writing tests for the unrefactored code.
The classification of testability refactorings is negotiable. It might be up to a developer's experience and style if he considers refactoring helpful for the creation of tests. When studying testability refactorings, it is important to clearly define what can be considered refactoring for testability. Is it just essential ones that are required for testing or is it also making testing easier, e.g., by making the source code more readable or less complex? It depends on the definition of testability that is used. When testability is seen as "the degree to which a component or system can be tested in isolation" [25, p. 1], understandability is not a relevant attribute because testing is possible even if it is more complex or takes more time. On the other hand, when the ease of testing a system is taken into account, understandability is important and more refactoring types like moving methods within the same block level or renaming methods are qualified for testability refactorings. The decision, of whether refactoring was done for testability, is not always trivial. The purpose of refactoring could be to improve the software quality, and a positive impact on testability could be a side effect.
To find refactorings in software projects, subsequent revisions of code need to be compared. For GitHub projects, these artifacts are commits and pull requests. Coelho et al.l studied the presence of refactorings in pull requests. In their sample of pull requests, they found 30.2% pull requests that induced refactorings, which appeared to be different from other pull requests in terms of the number of commits, comments and changed files. They also found that the majority of pull requests featuring refactorings had these

refactorings induced by code reviews. [20] Refactoring caused by review is unlikely to be for testability if not part of the review's result was missing tests. Even though this work did not focus on testability, the big amount of refactorings induced by pull requests is promising for finding refactorings for testability in commits and pull requests.
Sometimes it is necessary to perform multiple steps in the form of refactorings that are related to each other. These are called composite refactorings. Sousa et al. proposed heuristics to identify composite refactorings in commits. They found that most composite refactorings happen within the same commits and not across different ones. Further, they identified 111 composite refactoring patterns and found that it is likely that new code smells are introduced if composite refactorings are not done completely. [77]
Testability refactorings can also be composites. Reich and Maalej recently found ten composite testability refactoring patterns in an empirical study of pull requests on GitHub. [71] They were the first researchers to analyze how developers refactor code to create or modify unit tests at scale. Their research was done with Java projects and serves as inspiration for the work in this thesis.

3 Approach

This chapter will describe the methodology that is used in this thesis. Besides the analysis and evaluation of results, many tasks can and should be done in an automated way. Especially for tasks that need to be repeated for each entry of the dataset, automation is the key to efficiently getting the work done. Since this thesis deals with JavaScript, scripts and logic for analysis are written in JavaScript as well. The tools, e.g., for complexity analysis, parsing and AST traversal of JavaScript code, are often written in the same language. There is also a variety of libraries available, that can be easily integrated with package managers like NPM.
In addition to the need for scripts that can do some important tasks, there is a need for a comprehensive and clear interface to work with the data and keep track of the progress. Any data that is collected is stored in the JSON format since it is the standard way to work with data in JavaScript. Additionally, it has a readable and intuitive syntax. Data is locally stored in files, separated by their purpose.
Based on the requirements, a tool is implemented that uses a NodeJS backend to interact with the file system, communicate with external APIs and run analysis tasks. Additionally, a web-based UI is implemented. It communicates with the backend through a REST-API and enables a proper interaction with data. Using the Bootstrap[1] framework, UI components can be created with minimum effort. Besides some general HTML layouts, the tool consists of nearly only JavaScript code. For simplification, the tool is called JavaScript Testability Tool (JSTT) in the rest of the thesis. The features of the tool were implemented as they were needed and iteratively improved during the classification and analysis process because not all particularities of the analyzed projects were known in the beginning. Errors were carefully monitored during runtime so that the code could be adapted to minimize the risk of erroneous scripts.
The source code of JSTT and all the collected data are available in the replication package and on GitHub[2]. Setup instructions and more information on the tool as well as the collected data are available in the *README* file.

3.1 Sampling Strategy

The sampling of data for scientific studies is a part that is often treated with not enough attention. Since sampling essentially influences the quality of a study's results, it should

[1] `https://getbootstrap.com/`
[2] `https://github.com/Caramba997/JSTT`

be done carefully and follow some guidelines. Baltes and Ralph recently created a critical review of sampling in software engineering research. A main point of critique was that most studies are not representative, even if they claim to be. [12, p. 94]
This thesis will conduct an empirical study, so the selected sample should be as representative as possible. The first step is to identify the population of data. As already stated, this work will analyze JavaScript and TypeScript projects. Since the source code must be somehow available, only open-source projects are considered. For proprietary software, it is hard if not impossible to get a representative sample, because there likely exists no collection containing all or even a significant part of proprietary software. So the population is the entirety of JavaScript (and TypeScript) open-source projects.
There are some necessary restrictions to be made, to allow a feasible and efficient process of sampling the data. The first problem is that there exists no explicit data on how many JavaScript open-source projects exist in the world. They may be published on many different platforms or locations. There exists no complete list or collection of all projects. That is why the data source needs to be restricted. This thesis uses GitHub, the most popular platform for hosting open-source software, for sampling. There are other similar platforms like GitLab, Bitbucket or SourceForge, but considering all available platforms would be too much effort for the scope of this work. GitHub is a good choice because it is the most popular one and has a powerful API that allows you to search and query all the required data. Additionally, GitHub is frequently used in other similar studies. [11, 78, 1, 71, 79] GitHub holds hundreds of millions of projects, from which over 27 million are classified as JavaScript or TypeScript projects, not including forks.
The last restriction is to only consider projects rated with five or more stars, which is also used by Bogner and Merkel. [15] Projects with a low number of stars tend to also have low activity. [16] Other empirical studies, that used GitHub projects for their dataset also made some restrictions to exclude particularly small or irrelevant projects. Kochhar et al. only included projects with more than 500 Lines Of Code (LOC) to exclude ”toy projects” in their dataset. [53, p. 104] Especially for the second part of this thesis, a certain amount of development activity in the project is required. The restriction also reduces the risk of analyzing projects that are not used or not relevant anyway. Some studies put even more weight on the popularity of the projects and only use the top or at least highly rated ones. [27, 47, 76] But since the study's results should reflect the generality of projects, the lower star restriction is used. The first goal of this thesis is to find software metrics that correlate with test metrics, where low-quality code would lead to bad results. The fact that a repository has some stars does not enforce good quality but it is expected that it encourages a certain quality level.
No restrictions are made on the timeliness of repositories. Other studies sometimes only use projects that have recent activity, but testing is no new trend and projects without recent activity are expected to be relevant as well.
The resulting set of projects is the sampling frame of this work, JavaScript open-source

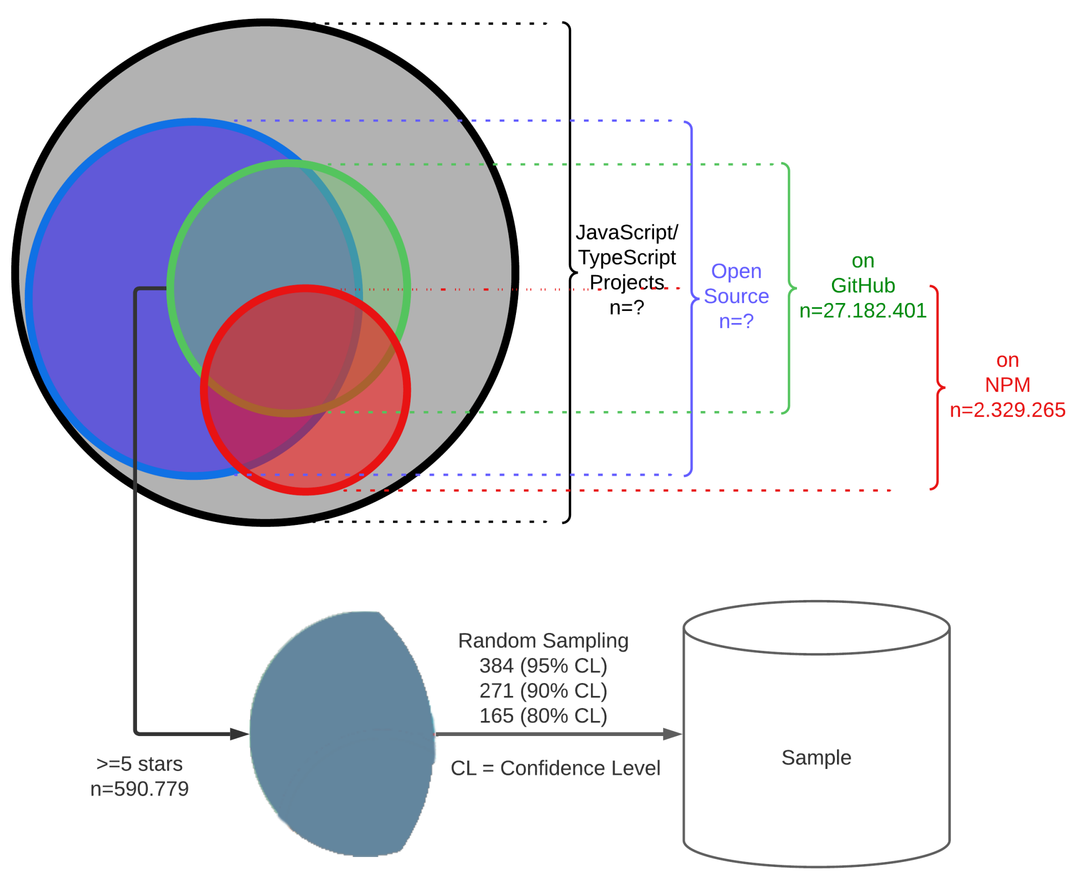


Figure 3.1: Sample creation

projects on GitHub with five or more stars. Simple random sampling is used to generate a sample that is representative of the sampling frame. Using an available online tool[3], the required sample size is calculated. An error margin of five percent and a confidence level of 95 percent is used, which is common to get meaningful results while having a handleable sample size. A lower error margin would drastically increase the required sample size. Figure 3.1 illustrates the sampling process with the restrictions made, as it is described. The set of projects on NPM is also represented because NPM was considered a possible data source as well. It was discarded because it holds way fewer projects than GitHub and the source code is often located on GitHub anyway. It is also expected that NPM projects focus more on reusability since they consist of packages that others can use, but other applications should not be excluded.

[3]e.g., `https://www.openepi.com/SampleSize/SSPropor.htm` or `https://www.calculator.net/sample-size-calculator.html`

3.2 Testability Analysis

To answer the first research question, the first part of this thesis will deal with collecting and analyzing software metrics to see if a connection can be drawn between test and source code metrics. Hypothetically, source code complexity metrics should correlate with test metrics. As explained in the fundamentals, there are principles and best practices to write testable code. In general, more complex code will be harder to test. That means a more complex module has more complex tests and test code. For example, if the cyclomatic complexity of a module or function is higher, one would need more test cases and more lines of test code to achieve a good branch coverage.
The analysis will build upon the dataset of projects that was described earlier. In a semi-automated process, different information will be collected from each project. Two categories of information are of interest. The first one is general information about the project like the field of use, application type, relevance and other metadata. Of special interest is the existence of tests, the type of tests (functional, performance, UI), and the use of test frameworks. Some types of tests may occur not very frequently. UI tests will of course only occur when a User Interace is present. Performance tests are usually not implemented as often as unit tests. Security tests will most probably not be reflected in the project's codebase, but this hypothesis will be confirmed or disproved after the analysis.
The second category of information comprises the software metrics. For each project and its corresponding source code and test files, as many of the predefined metrics as possible are collected. Most metrics will be static ones, so it should be possible to calculate them for most of the files. Since in general, dynamic metrics can only be collected when it is possible to run the test or program, they will be collected whenever possible. The only dynamic metrics that will be collected correspond to testing, so the tests need to be executed. A reasonable amount of effort is made to get the tests up and running to collect the metrics of interest. Running tests or software can be a challenging task due to preconditions, software dependencies or insufficient documentation regarding instructions for setup and installation. As a consequence, this is not always possible.
Since a big set of projects should be analyzed, the collection of metrics should be automated as much as possible. Nevertheless, the process should be monitored closely to enable manual actions in error scenarios. Due to varying file structures, programming styles and other peculiarities of the projects, the automated workflows may have wrong mappings or results, so it needs to be possible to inspect results easily and check for or correct faults. JSTT provides a comprehensive user interface for working with the data and starting the scripts.
An important task that is done with JSTT's UI is the establishing of connections between test and source code files. For correlation analysis, pairs of metrics are needed. The task consists of creating one-to-one mappings for the files, which is not always possible, so

source code files can appear in multiple connections. This is the case when multiple test files test the same module, e.g., when a project uses big bundled files or the tests are integration tests.
Once all projects are processed, the correlations between test and source code metrics are computed. The results and the set of metrics can be compared to findings in other studies, even if they target other languages like Java. It will be interesting to see possible similarities or differences. The set of metrics will also be used to develop a scoring system that allows assessing the level of testability of source code. The scoring system is applied to the source code in the sample and the results are evaluated to find out which factors lead to better or worse testability.

3.3 Selection of Testability Metrics

As already mentioned, most studies on analyzing testability focused on Java applications. In section 2.6, metrics for testability measurement collected from the literature were described, and table 2.1 summarizes the findings. A selection of them will be used in this thesis. Metrics that apply to class-based software architecture and can not be adapted to JavaScript's script nature are discarded. An example of this is Depth of Inheritance Tree (DIT) because even though there is syntax to inherit from classes in JavaScript, the many possibilities in dynamic prototype chaining make it very hard to determine the DIT. Metrics regarding the visibility of class members like Percent Public And Protected (PAP), Number of Public Methods (NPM) and Number of Private Methods (NPRIM) are also problematic in JavaScript. As explained in section 2.3, it is possible to create public and private class members, but the visibility of methods or variables, in general, can be changed dynamically. Those metrics would be suitable if only TypeScript projects were analyzed.
When there was no tool found to collect metrics for JavaScript and it is unlikely that a custom script can reliably compute them, they are discarded as well. An example of this is Lack of Cohesion in Methods (LCOM). Another example is Number of Bytecode Instructions (NBI) because JavaScript is an interpreted language and different engines produce different bytecode. The remaining metrics form a basis for this work. They are extended with additional complexity metrics that the used analysis tools output anyway, like the Halstead metrics.
After the collection of metrics from all the projects, a correlation analysis is done between test and source code metrics, as it is commonly practiced in the literature. [2, 10, 11, 18, 30] The reason for the correlation analysis is that the testability of the source code is expected to be reflected in some metrics. The testability of the source code is also expected to influence aspects of the test code, which are reflected in metrics again. If a strong correlation is found between a source and a test code metric, the source code metric influences the test code and is therefore connected to testability.

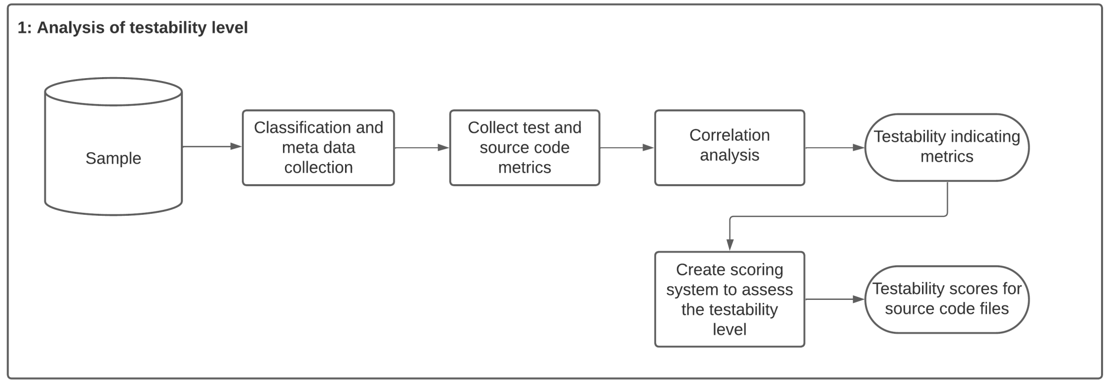


Figure 3.2: Approach to find testability metrics

The outcome of the correlation analysis will be used to give some advice on how to assess the testability of JavaScript code. The metrics that have a significant correlation to test measures are likely to be important for testability if the corresponding test metrics are found to be relevant. As an example, coverage test metrics are interesting because they are good indicators of the performance of test suites. Also, test complexity metrics can show the ease of testing. It will be discussed if the results make sense in the matter of testability indication. Limitations on this will be discussed in the end. Figure 3.2 illustrates the process of finding source code metrics related to testability.
In the following, the selection of metrics to be calculated is explained. They will later be evaluated and only some will be used in the evaluation. At this point, a large set of metrics is beneficial and unnecessary metrics can just be discarded later on. The tables 3.1 and 3.2 show an overview of all metrics. Table 3.1 shows the metrics related to the source code and table 3.2 shows the metrics related to the test code.
Metrics can belong to two different scopes. The first is module scope (M), where a module is a file of code. The second is function scope (F). In the course of this thesis, those letters are appended to the metric names to indicate the scope. E.g., *locM* means module scope and *locF* means function scope. Many metrics can be computed for both scopes. Since JavaScript is a scripting language and not class-based, a file/module may contain code that is not wrapped in functions. For this reason, function metrics can not always be computed, but generally, it is interesting to calculate metrics for both scopes. Since a single value is wanted for each metric, function scope metrics are calculated as aggregate values per module. The sum (total), average (avg), median (med), minimum (min) and maximum (max) are computed for further use. In the course of this thesis, the short forms in brackets are appended to the metric names to indicate the aggregate. E.g., *ccF_total* means the sum of cyclomatic complexities for all functions of a module.
The resulting set of metrics holds a total of 24 metrics on module scope and 75 metrics on function scope. Six of the metrics on module scope apply only to the test code and

the rest applies to both source and test code.

LOC

Lines Of Code (LOC) is a generally interesting and easy computable static metric for the size of software. Additionally, it was often found to be correlated to testability. It counts the number of physical lines of code in a module/function.

LOCC

Lines Of Code Comments (LOCC) counts the lines of code comments in a module. Documentation of software affects readability and understandability, so a higher amount of comments may lead to higher testability.

LOCL

Logical Lines Of Code (LOCL) is a variation of the LOC metric. Since LOC highly depends on the code style, this metric can give a more generalized view of the actual size of the code.

NOF

Since JavaScript is not class-based, metrics like Number of Classes (NOC) or Number of Methods (NOM) do not directly apply but can be adapted. Number Of Functions (NOF) is another size metric and states the number of functions in a module. Since a file is treated as a module and metrics are calculated per file, a metric for the number of modules makes no sense. In JavaScript, the source code is often separated into different files to get a similar separation of concerns like one would do with classes.

CC

Cyclomatic Complexity (CC) is a popular complexity metric, that counts the number of individual paths through a piece of code, which can be applied to different scopes. A higher value means more complex code and - for source code - indicates the need for more test cases to cover all individual paths.

Halstead metrics

Halstead metrics are complexity metrics that are based on separating the source code into operators and operands. Derived from complexity, some metrics aim to provide estimates for the effort for testing and comprehension. [42] In the following, the used Halstead metrics are briefly explained.

Halstead Length (HL) is the total number of operators and operands.
Halstead Vocabulary (HVOC) is the number of unique operators and operands.
Halstead Difficulty (HD) is the ratio of unique operators to the total number of operators.
Halstead Volume (HVOL) is the product of HL and the logarithm of HVOC.
Halstead Bugs (HB) is the expected number of bugs.
Halstead Effort (HE) is the effort required to understand or implement a piece of code and is the product of HD and HVOL.
Halstead Time (HT) is the estimated time to write a piece of code.

NOP

Number Of Parameters (NOP) is the number of parameters of a module.

MI

Maintainability Index (MI) is a composite of other metrics and gives an estimation of how well-maintainable a module/function is. Its components are HVOL, CC, LOC and the ratio of comments to LOC.

EC

Efferent Coupling (EC) is also called import coupling and states the number of imported modules.

AC

Afferent Coupling (AC) is also called export coupling and states the number of modules that import the given module.

ND

Nesting Depth (ND) is the maximum level of nested blocks.

MC

Method Calls (MC) is the number of method calls in the module.

Coverage metrics

Coverage metrics can indicate the effectiveness of test suites by stating how much and which parts of the underlying source code is executed when running the tests. Different scopes can be considered.

Line Coverage (LCOV) states the percentage of lines that are executed.
Statement Coverage (SCOV) states the percentage of statements that are executed.
Function Coverage (FCOV) states the percentage of functions that are executed.
Branch Coverage (BCOV) states the percentage of branches that are executed.

NOTC

Number Of Test Cases (NOTC) is the total number of test cases in a test file.

TLOCR

The ratio of TLOC and LOC is a good indicator of how well a system is tested, meaning a higher number of LOC usually leads to a higher number of TLOC. Ideally, the ratio of test and production code should be 1:1, so a value smaller than 1 can reveal missing tests. [2, p. 2] Fard and Mesbah used this as a quality metric for tests. [27]

Table 3.1: Source code testability metrics

Metric	Description	Scope
LOC	Lines Of Code	M, F
LOCC	Lines Of Code Comments	M
LOCL	Logical Lines Of Code	M, F
NOF	Number Of Functions	F
CC	Cyclomatic Complexity	M, F
HL	Halstead Length	M, F
HVOC	Halstead Vocabulary	M, F
HD	Halstead Difficulty	M, F
HVOL	Halstead Volume	M, F
HB	Halstead Bugs	M, F
HE	Halstead Effort	M, F
HT	Halstead Time	M, F
NOP	Number Of Parameters	M
MI	Maintainability Index	M
EC	Efferent Coupling	M
AC	Afferent Coupling	M
ND	Nesting Depth	M, F
MC	Method Calls	M

Table 3.2: Additional test code testability metrics

Metric	Description	Scope
LCOV	Line Coverage	M
SCOV	Statement Coverage	M
FCOV	Function Coverage	M
BCOV	Branch Coverage	M
NOTC	Number Of Test Cases	M
LOCR	Test Lines Of Code to Lines Of Code Ratio	M

3.4 Finding Refactorings

Code does not always allow for proper testing in the first place. Tests are induced or improved over time and may require changes in the source code as well. Those changes can be called refactorings for testability. After studying the presence of tests in JavaScript projects, Fard and Mesbah proposed to study ”refactoring techniques towards making the code more testable and maintainable”. [27, p.239]. Reich and Maalej already analyzed refactorings for testability written in the Java language. [71]
To find out how developers improve their code to make it more testable, the dataset with the additional information gathered to answer the first research question can be used. The sample is used to retrieve commits and pull requests. More precisely, the repositories that were found to have tests are used as a starting point to search for commits that modify pairs of source and test files. The goal is to find refactorings in the source code that were needed to allow the creation or improvements of test code. Refactorings that improve the code quality in aspects like readability, maintainability or modularity in general could often also be seen as improvements in testability, but only those that are done to allow better testing are considered as refactorings for testability at this point. So it is the purpose of the refactoring that matters. Otherwise, it would be difficult to draw a line between the refactorings during the classification.
To find refactorings, the source and test code need to be somehow related. The connections between the source and test files are already established in the first part of the analysis. It is now possible to search for commits that modify the pairs of files. Some of the resulting commits are expected to contain refactorings for testability.
Another approach that is used by Reich and Maalej [71] utilizes keywords to identify relevant pull requests for finding refactorings for testability. In the following, this approach will be called the keyword approach. Using keywords is a common practice [3, 20] to find commits of interest since it is easy and does not require a lot of additional work. Nevertheless, keywords do not guarantee that all relevant results are found because they assume developers documented their intentions carefully in their commit messages. And even if the refactorings are documented, there is a huge variety of terms

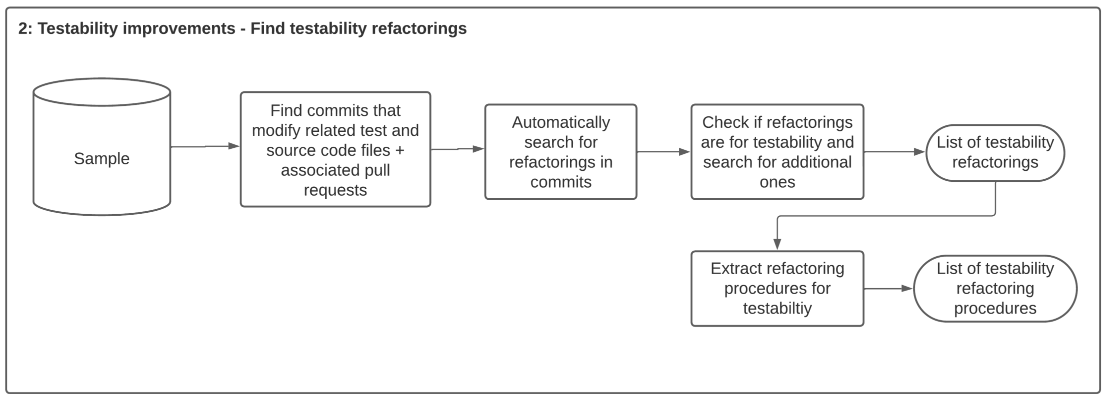


Figure 3.3: Approach to find testability refactorings

that are used to describe them. Therefore, besides the keyword-generated sample, Reich and Maalej also used a random sample for comparison. [71] AlOmar et al. iteratively studied commit messages and found a list of 87 patterns used to indicate refactorings. [4] This long list shows the difficulty of using textual descriptions to find the commits of interest.
The goal of this thesis is not only to find refactorings but those that improve testability. Due to the expected difficulty of achieving a high recall with keywords and the fact that a dataset of handleable size exists, all commits that contain changes in both source and test files will be used to find refactorings for testability. Reich and Maalej used the keyword approach because they expected the approach proposed here to fail in giving meaningful results. It is an interesting question how the two approaches perform in comparison.
This strategy also led to the decision to use commits instead of pull requests. Pull requests consist of at least one commit and may better describe the changes that were made. But especially small repositories that do not have many contributors do not necessarily make use of pull requests. The dataset does not only hold big projects that are likely to use the pull-based development, it is expected that the use of only pull requests would lead to missing relevant refactorings for testability. Also, the textual descriptions of pull requests are not needed to identify the relevant ones and pull requests are just wrappers with additional meta information around commits. It is expected that co-changes in source and test code are not distributed around different commits in the same pull request, since refactorings that are needed to write, improve or extend tests will be needed just at this moment and not beforehand or afterward. Commits also have equally or fewer changes than corresponding pull requests which makes an analysis more manageable.
In the resulting commits, refactorings are identified automatically with existing refactoring mining tools. These tools are rare for JavaScript, it appears that most tools focus on other languages like Java. The commits are therefore always reviewed manually too to find missing refactorings. For every mined refactoring it is determined if it improves

testability. Additionally, the source and test code are analyzed to find missed refactorings for testability. Figure 3.3 illustrates the process to answer the second research question.

4 Analysis

4.1 Dataset

As described in section 3.1, the sample is a random selection of GitHub repositories that are written in JavaScript or TypeScript and rated with five or more stars. A workflow containing three elementary steps was developed to create the sample. Each step is automated and can be executed with JSTT.

In a preliminary step, the exact total number of repositories matching the criteria is retrieved. GitHub has an extensive REST-API that allows searching for various types of resources. The approach was to conduct a search query that includes the criteria and then see how many results the GitHub API returns. Since only the total number of repositories is of interest, it is sufficient to make the query return only the first match. The API returns the total number of results matching the query separately.

There is one issue with this process that has to be worked around. GitHub has restrictions on how many resources and how much time an API request can take, so the desired total number can not be determined by one single search request. Without further query restrictions, the API response indicates that the results are incomplete. Because of this, a way to split the data into well-defined parts is needed. A solution is to group repositories by creation date. It is queried again for every year separately starting with the launch of the GitHub platform and ending with the year this study was conducted (2023). When the response still indicates incomplete results, the period of one year is split into three parts with four months each, which was found to be sufficient to always deliver complete results. This process is done for JavaScript and TypeScript separately since directly including both languages would increase the complexity of the query and the periods would be needed to be even smaller. Besides, it was interesting to know how many repositories exist for each language. The total number of repositories for each period and language were accumulated. The result was that GitHub holds a total of 593,813 repositories matching the criteria, with 79% (470,561) being JavaScript and 21% (123,252) being TypeScript. Derived from the total number of repositories, the required sample size was found to be 384.

The second step is to generate some random numbers for a random selection of repositories for analysis. The randomness comes from truly random numbers generated with random.org[1]. The numbers are in the range of one to the total number of repositories.

The third step is fetching the repositories. A similar approach to step one is used to map

[1] `https://www.random.org/`

each random number to a repository. The results from step one, which are periods with a corresponding number of repositories, are reused here. For each random number, it is checked in which period it belongs. A period holds numbers starting from the total number of repositories in all previous periods plus one and ending with the start number plus the total number of repositories in the period. This way, it can be identified in which period to search for the repository. Due to restrictions of the GitHub API, the period needs to be split again in most cases. The API allows you to retrieve the first 1000 results for a query, even if you only want one match. The results are paginated, but GitHub returns no result if you request a page that is outside the limit. If the random number minus the period starting number is bigger than 1000, the period is split in half. Out of the two resulting periods, the one that has its ending number greater or equal to the random number is used as the new period used to search for the repository. This divide-and-conquer approach is repeated until the repository can be retrieved. Every resulting period from this process with its information about the total number of repositories is cached for reuse on the remaining randoms.
Unfortunately, the repositories returned by a search query can not be ordered by the creation date. That means that in the last iteration of the divide-and-conquer process, not the exact repository that would be chronologically at the desired position is returned. The order of the results for a query depends on GitHub's ranking algorithm for the best-matching repositories. Since no search term is specified in the query, it is not completely clear, how best matches are determined by the GitHub API. Nonetheless, the process is expected to be accurate enough to deliver a random sample. The alternative would be to retrieve all repository data for a period that holds 1000 or fewer repositories and order them by creation date manually. This way, the process would take significantly more resources concerning storage space and time.
In the end, a dataset with 384 distinct repositories is created as the basis of this work.

4.1.1 Meta Data

Before it comes to the collection of metrics, some general insights into the dataset are retrieved. The ratio of JavaScript to TypeScript projects is approximately the same as in the sampling frame, only differing in the decimals of the percentage. Of the 384 repositories, 303 are classified as JavaScript and 81 as TypeScript. GitHub uses a third-party library to determine the languages of a project's files. That is not always correct, as the dataset shows. In general, projects often contain not only code of one language but a total of 5 projects that were classified as JavaScript projects did not contain any JavaScript code. They just contain JSON or configuration files that are written in JavaScript but only used to configure some libraries.
Some projects have files containing JavaScript code but with special file endings, they are not seen as misclassifications. 2 projects use the file extension *.gs*, which stands for

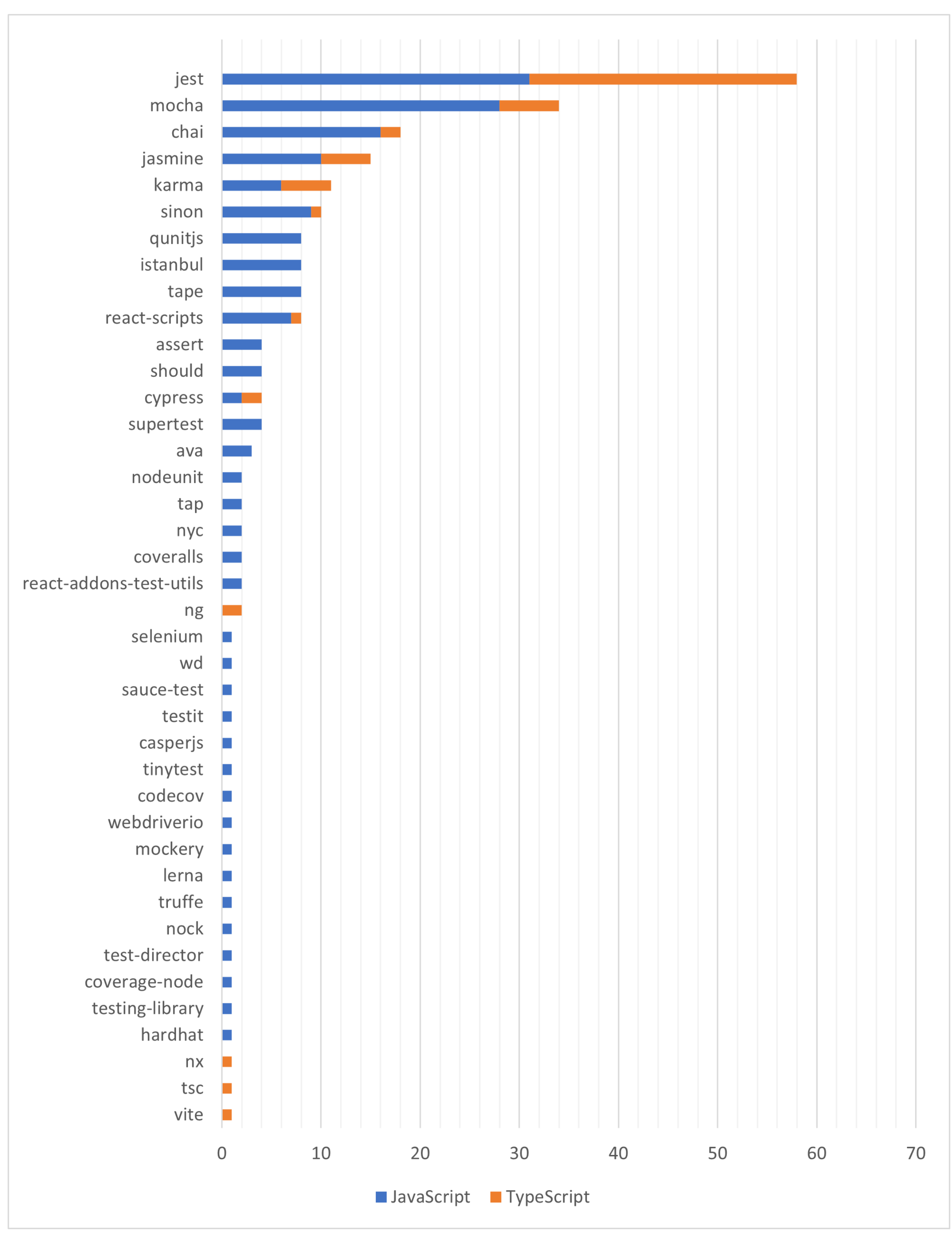


Figure 4.1: Test frameworks

Google Apps Script and is basically JavaScript. One project uses the file extension *.pac*, which stands for *Proxy Auto-Config* and contains a JavaScript function.
Due to the misclassifications, the actual total number of JavaScript projects is 298. In the following, all numbers refer to the total number without misclassifications (Total: 379, JavaScript: 298). Percentage numbers are rounded.
91 repositories are owned by organizations and 288 repositories are owned by individual GitHub users. The star rating ranges from five to 34,429 with an average of 245.8 but a median of 21, indicating that most repositories are not very highly rated. Only one repository, which is JavaScript, has way above 10,000 stars, the second most-rated one has 4,104 stars. Disregarding the most popular repository, the ratings are similar for JavaScript and TypeScript with the same median and slightly different averages (Median: 21, Average: JS 150 & TS 175).
It was checked how many repositories use NPM because the usage of the package manager makes it easy to check for dependencies or the prevalence of tests by searching for scripts that trigger test runs. The identification is based on whether the repository contains a package.json file or not. This action was done using the GitHub API by searching for the file in the repository but validated in the manual reviews which will be explained later. It came out that 290 repositories (77%) use NPM. Further, while only 72% of JavaScript projects used NPM, all but two TypeScript projects did.
As a measure of the development activity in projects, the total number of commits and pull requests for each repository were retrieved using the GitHub API. The dataset holds 57,349 commits and 17,359 pull requests in total. It appears to be that TypeScript has significantly more activity than JavaScript projects. For both languages, the minimum number of commits is 1 and the minimum number of pull requests is 0. Average and median values in contrast are higher for TypeScript regarding commits (average +45% and median +89%). Regarding pull requests, the difference is even more significant (average +202% and median +350%).

4.1.2 Manual Classification

Some information about the projects in the dataset is difficult to obtain automatically. That is why some information is gathered manually for each repository. In the following, all manually collected data and its results are explained.
First, it is checked if the project contains frontend or backend code to be able to compare the results later. Frontend code is seen as code that interacts with a graphical user interface, most often with a DOM. Backend code is seen as code that contains server logic. It is also possible that a project has no frontend or backend code, which means the execution environment is not determined. E.g., the code could consist of scripts that can run anywhere to do some calculations for some input value or it could be an API connector that could either be used on a server or locally in a script. Projects that contain

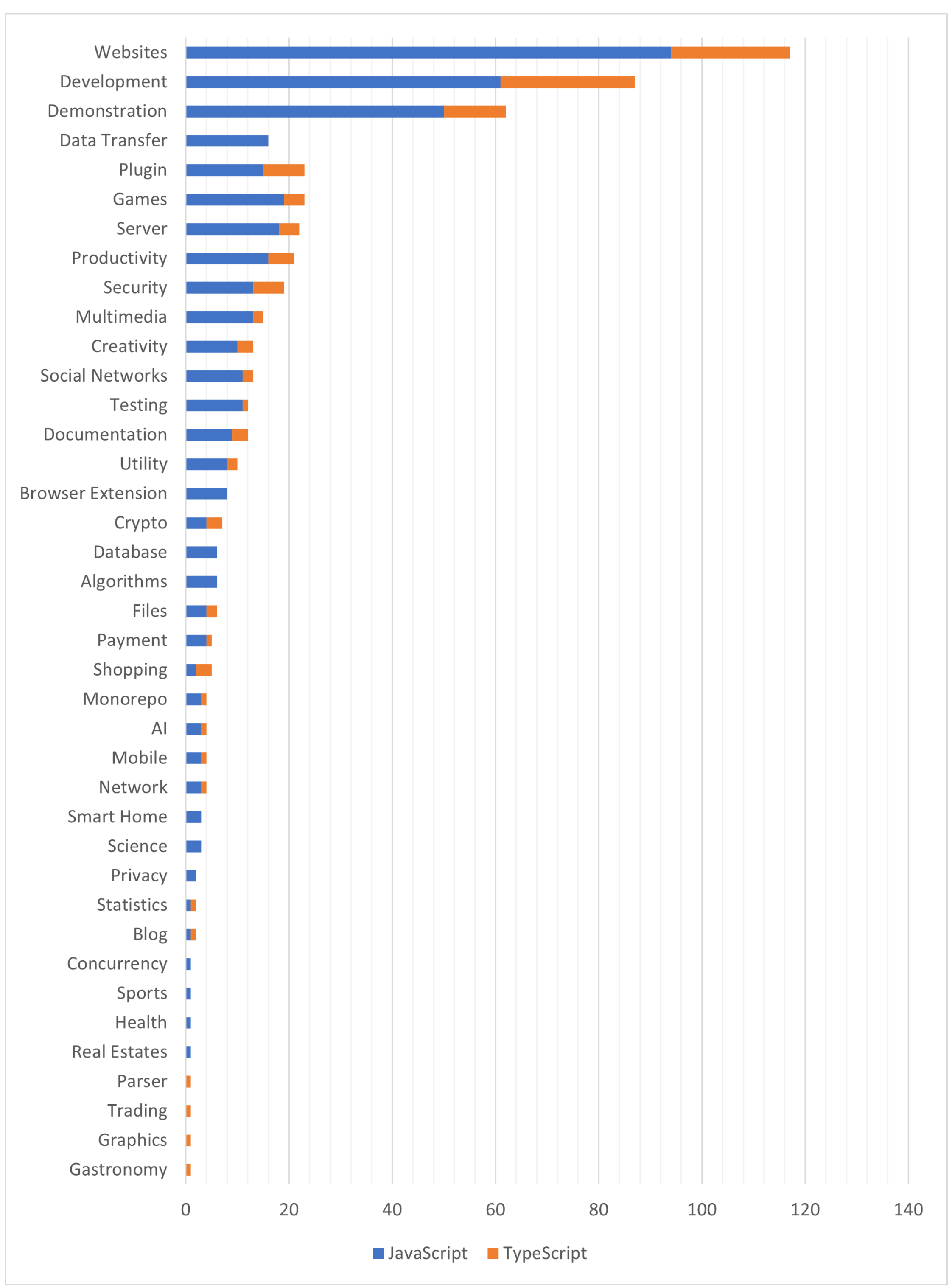


Figure 4.2: Project categories

frontend and backend logic often use a UI framework like React. The dataset contains 210 projects with frontend and 131 projects with backend logic. For TypeScript, there are proportionally slightly more projects with backend code than for JavaScript.
Probably most important, for each project it is evaluated if it contains tests or not. A project is classified as having tests if at least one test case is found. In some projects, the only existing test cases are commented out or written in another language than JavaScript or TypeScript, in these cases, the projects are classified as not having tests. If a project has tests, it is further evaluated if unit tests, UI tests and performance tests are present. In total, 148 projects (39%) had tests. Interestingly, this number matches the expected amount of tested GitHub projects written in Java estimated by Madeja et al. [57] Still, the number differs significantly from the 78% of JavaScript projects having tests in the study by Fard and Mesbah [27, p. 234]. They used a different dataset that focuses on more popular JavaScript projects, making it clear that the selection of the sampling frame heavily influences the presence of tests. In this sample, with 48%, TypeScript projects appear to be more tested than JavaScript projects with only 37%. 50 of the 210 projects having frontend code are implementing at least one UI test case that tests interactions with or the correct rendering of UI components. Only 5 projects contain performance tests.
In nearly all projects, at least one framework is used for testing. The most frequently used test framework is Jest, followed by Mocha, Chai and Jasmine. Figure 4.1 shows all identified test frameworks with their frequency, also for JavaScript and TypeScript separately.
46 of the projects have textual artifacts not written in English, which sometimes made it harder to determine their purpose. Most non-English projects are Chinese. For every project, the categories it belongs to were examined. A project can belong to more than one category. As an example, a project that is used for demonstration often is assigned to another category that says what is demonstrated. Figure 4.2 shows how many projects there are in each category. Approximately 30% have code that is related to websites, which could be custom functionality or a plugin or library to use on a website. 23% of the projects contain code that is related to software development. That could be tooling or libraries for certain tasks or demonstrations on how something can be done. The third biggest category holds projects that demonstrate something. As already said, the demonstration could be related to development but it could also be a project to experiment with something. The data transfer category holds projects that range from implementations of network protocols to libraries that do some data exchange with external APIs.

4.2 Testability in JavaScript-Projects

This section will describe the process of answering the first research question in detail. It explains, how every step of the process is done and which results occurred.

4.2.1 Collection of Metrics

The goal is to calculate correlations between test and source code metrics. For this reason, three pieces of information are needed. The first two are test and source code metrics for each file of the corresponding type. Also, the connections of test files to the source files under test are needed. For correlation analysis, there has to be a one-to-one mapping of metrics. It is not always possible to map one test file to exactly one source file, in this case, the connection is omitted to not impact the results negatively.
With the help of JSTT, every project that has test cases is analyzed sequentially. There are scripts to extract static metrics from source and test files automatically. Test files are identified by keywords in their paths. It is looked for either *test*, *spec* or *cypress*, which appeared to be sufficient for most cases. The last keyword is used, because when the Cypress testing framework is used, files may be located in a *cypress* folder. The *test* keyword was found to be used very frequently throughout projects, even as part of the filename or as the name of a folder. The *spec* keyword is also frequently used to identify test files in JavaScript projects. Kochhar et al. also used the *test* keyword when analyzing projects of different languages for the presence of tests. [53] Madeja et al. used the *test* keyword to find test files and analyze the correlation of the usage of the keyword within a file with the number of test cases in Java. [56, 57] For JavaScript, it appeared to be insufficient to use the *test* keyword alone. Since every project is reviewed manually after the automatic evaluation, the final identification of test files should be fairly accurate. In some cases, files were classified falsely by JSTT due to many different styles of structuring projects and naming resources, they were manually corrected. For identifying performance tests, the keywords *bench* and *perf* were used.
The mapping of test files to source files was also done automatically by comparing the file names and paths and checking the imports in test files. For some projects, this worked perfectly but for others, it did not. So the mapping was reviewed and adjusted manually. For performance tests, the mapping was done manually because the effort for the few projects that had performance tests was moderate. Metrics were calculated for performance test files like they were for unit tests. It turned out that only four repositories had performance tests.
To calculate the static metrics, a variation of the *ESComplex* library is used, which computes many complexity metrics given a piece of code. There are not many libraries available for metric collection in JavaScript, even this one is not very actively maintained anymore. But it appeared to still work as desired. Coupling metrics, nesting depth, number of method calls and lines of code comments are computed by custom algorithms operating on either the AST or the string representation of code. It was a challenge to support all possible formats of code, which were mostly JS, TS and React's versions of JS (JSX) and TS (TSX). For JavaScript, there exist some files with different file extensions that can be treated as usual JS. *ESComplex* can, when configured correctly, process all needed formats. The right plugins need to be set up beforehand,

depending on the file type. *ESComplex* uses the *Babel* parser to generate the ASTs of code, which is also used for the custom metric computation outside of *ESComplex*.
One additional metric, the ratio of test to source code, could be computed from the existing metrics after the connections between test and source code were established. As explained in section 3.3, the ratio should be 1:1. For this reason, the metric was expected to be a good indicator of testability.
As expected, none of the projects had implemented security tests. They may be using them but it is not documented, so no conclusions can be drawn here.
For every project with test cases, some additional metrics needed to be collected manually. The number of test cases could not be reliably computed automatically, because depending on the testing framework used or even the programming style, the methods or ways used to implement a test case vary. Often, test frameworks define methods named *it* or *test* to let you write test cases, but there are different ways like defining test cases in arrays and looping through them afterward.
A relatively large effort was put into collecting coverage metrics for tests. The amounts of statements, branches, functions and lines of source code that get executed when running tests were expected to be a promising measure for the testability of the source code. It was not always possible to execute tests or compute the coverage. There are many different reasons for that. Especially for old projects that were not updated for a long time, dependencies did not exist or were not working anymore. A problem was missing or insufficient documentation about the prerequisites to run the test suites too. It was often not clear how the environment needs to be set up when software depends on it. This study was done on Windows, most probably many projects were written on Linux systems. Even projects were found that documented they could only run on Apple systems. Some things like build tasks errored on Windows even if they should function because prerequisites were installed. E.g., some build tasks needed Python and other software installed, but it was difficult to get the right versions and dependencies, especially for old projects. Some projects did have test files but with only boilerplate code that was not properly implemented or executable.
From the 148 projects having tests, for 45 projects the tests could not be executed. For another 32 projects, some or all tests were failing. The reasons for failing tests were partly the same as for the not executable ones. Dependencies were outdated, the environment was missing something and some tests failed because of incorrect code. For both the tests that executed correctly and those that produced failures due to apparent code issues, attempts were made to produce coverage metrics. There is one popular tool for coverage computation, that is used commonly and supports most testing frameworks. It is called *nyc* and is a command line interface for the *Istanbul* engine. It instruments the source code and outputs a summary at the end. Depending on the testing framework, there may be some configuration and setup needed to calculate coverage metrics. For some UI tests, the code needed to be instrumented beforehand and the coverage needed

to be collected from the browser's console. There were many different setups required to get as many coverage reports as possible. Finally, coverage could be computed for 434 source code files in 74 projects. In general, 843 pairs of test and source code metrics from 139 projects could be established.

4.2.2 Correlation Analysis

Correlations are computed using Spearman's Rank Coefficient, which is a common practice in similar research. [2, 9, 10, 11, 18, 54, 78] It is also referred to as Spearman's RHO. The advantage of using Spearman over Pearson's correlation coefficient, another famous method of determining correlations between two datasets, is that it does not require the relationship to be linear, any monotonic correlation is recognized by Spearman's method. That is achieved by not using absolute values but ranks for each piece of data. The rank coefficient is a value between -1 and 1, while values close to 0 indicate there is no correlation. A negative value means that one variable increases while the other one decreases. A positive value means that both variables increase or decrease. Which values indicate a significant correlation depends on the author you choose. For Xiao et al. it is an absolute value of the correlation coefficient greater than 0.5. [87] This limit is also used in other studies [2, 78], while some use even lower limits for significant correlations [9, 18, 54]. An absolute correlation coefficient between 0.3 and 0.5 indicates a moderate correlation, everything below indicates none or a weak correlation. [87]
In addition to the correlation coefficient, the p-value is computed for each relation, which indicates if the result is significant. P-values give the probability that the null hypothesis can be discarded, with the null hypothesis stating that the calculated correlation did appear randomly. Only when the p-value is low, the correlation coefficient is significant. The limit for an acceptable p-value depends on the field of study, often 0.05 or 0.01 is used. [87]
Unfortunately, existing libraries for JavaScript to calculate the correlation coefficient were not giving satisfying results. They either returned false results, were unable to give any results or were missing the p-values. For this reason, a little Python API was implemented that only calculates correlation coefficients and corresponding p-values and returns them to JSTT. Python seems to be more extensively used for statistics and was found to have better and more reliable libraries in this field.
Correlations are calculated for each possible pair of test and source code metrics. The result is that 43 correlations are strong with absolute correlation coefficients ranging from 0.5042 to 0.6539. Table 4.3 shows the correlation coefficients, for better readability only strong correlations are displayed. 1310 pairs of metrics are moderately correlating, but only the strong correlations are used for further evaluation. Since the dataset of metrics is relatively large, the p-values for all strong correlations are very low (between $1e{-}19$ and $1e{-}65$) and allow discarding the null hypothesis and considering the correlations

as significant. The branch coverage metric *bcovM* appears to be the most meaningful test metric since it negatively correlates with 33 source complexity metrics. The source metric *dpM*, which is the maximum nesting depth of a module, negatively correlates with the line coverage test metric *lcovM*. A conclusion from this is that more complex modules lead to lower test coverage and thus to lower testability.
The efferent coupling of a module *ecM* correlates with *ecM* for the associated test file. It is the only positive strong correlation. It is also the only encapsulation source code metric with a strong correlation. It may indicate that tests for modules with many dependencies need those dependencies too. During the classification phase, it could be noticed that it depends on the used testing framework which dependencies are needed. Sometimes, tests do not have a single dependency. The test framework loads itself during runtime and the module under test is imported automatically by naming conventions. The test file *test.spec.js*[2] in the *LancerComet/lens.js* repo is an example that only imports the module under test. Other test files import multiple testing libraries and also libraries to set up the local test environment and load fixtures, e.g. *generator_spec.js*[3] in the *mustardamus/lehm* repository. The associated source file has an even higher number of imports, but the imports differ from the ones in the test file.
The last test metric that appears in the strong correlations is *locrM*, the ratio of test code to source code. Since it is calculated using the *locM* metric of the source code, the strong correlations to *locM* and *loclM* are obvious and meaningless. *locrM* correlates with most Halstead complexity metrics on module scopes. That also holds for *bcovM*, which additionally correlates to the Halstead metrics on function scope. The Halstead metrics appear in about 60% of the strong correlations and seem to be a good indicator of testability.
For the source metrics on function scope, aggregate values were used. Only the total and max aggregates appeared in the strong correlations. When both aggregates of the same metric appear, it is always the total that has a higher correlation coefficient. Contrary to the maximum, the total takes all functions into account and may better represent the module as a whole. However, a complex function in a module seems to lead to lower branch coverage as well.
For every one of the complexity metrics in the strong correlations, a higher value means higher complexity, so it goes along with the guidelines and advice to write testable code that the coverage gets higher when the complexity gets lower. Interestingly, no test complexity metric other than *ecM* strongly correlates with source code metrics. That means no significant connection between the complexity of the test and the source code could be drawn. Instead, it is the test coverage that connects the quality of tests to the quality of the source code. A problem is that it is not guaranteed that the developers of the analyzed tests aimed for maximum test coverage. If they did but the coverage is low,

[2]`https://github.com/LancerComet/lens.js/blob/master/test/specs/test.spec.js`
[3]`https://github.com/mustardamus/lehm/blob/master/test/generator_spec.js`

one could conclude that the testability is low because it was not possible to achieve a high test coverage. This is a general problem that will be further discussed in the threats to validity.

4.2.3 Testability Level Analysis

In summary, 35 distinct source code metrics strongly correlate with test metrics. Using this set of metrics, the repositories in the dataset are compared regarding their level of testability. A rank-based scoring system is proposed to assess the level of testability for a file. The goal is not to find an absolute value that shall assess the testability but to compare the files. It would require further studying to find the optimal values for each metric regarding testability if they even exist. A rank-based approach, like it was used to calculate the correlation coefficients in Spearman's RHO, allows comparing metric values independent of the distribution of the value space. For all complexity metrics in the set, a lower value often leads to better testing, so lower values are considered to lead to better testability. All 35 metrics are equally weighted.
Before explaining the scoring system, some limitations need to be mentioned. As already stated in section 2.5, testability is a complex software attribute and can involve many different aspects. The metrics that are used here are mostly complexity metrics and miss other important characteristics. E.g., the visibility of methods and attributes is very important for testability, because a method with low complexity can still not be tested when it is not accessible in tests. When counting understandability as part of testability, code comments, formatting and naming are also important aspects of source code because they can heavily influence the effort a developer has to understand a module and write tests for it. To conclude, the scoring system only addresses one important aspect of testability, which is code complexity.
The scoring system is built on the metrics calculated for all source code files in the used dataset but can be applied to any other file afterward. In the following, the process is explained. First, all existing values from every analyzed source code file for the 35 testability-related metrics are collected in sets, meaning only distinct values are kept. These sets are sorted in ascending order. Now, for every source code file the ranks for each metric are derived. This is done by checking the index of each value in the sorted sets. A lower rank means a better metric value and therefore better testability. The rank equals the index plus one to make the ranks start at one. Files that have equal values for a metric also have the same rank for this metric.
The ranks are normalized to the range of zero and 100 to guarantee an equal weight of every metric in the score. To get one single rank value for each file, the average rank of all available metrics is computed. This allows ranking a file even if it misses some of the values for the 35 metrics. It appeared that this was the case for some files, which mostly missed the metrics on function scope. JavaScript as a scripting language does not

Source code metric	Test metric: lcovM	bcovM	ecM	locrM
dpM	-0.5042			
locF_total		-0.6028		
locF_max		-0.5364		
loclF_total		-0.5904		
ccF_total		-0.5980		
ccF_max		-0.5629		
hbF_total		-0.5977		
hbF_max		-0.5155		
hdF_total		-0.6221		
hdF_max		-0.5666		
heF_total		-0.5885		
heF_max		-0.5504		
hlF_total		-0.6047		
hlF_max		-0.5235		
htF_total		-0.5885		
htF_max		-0.5504		
hvocF_total		-0.5845		
hvolF_total		-0.5975		
hvolF_max		-0.5155		
nopF_total		-0.5425		
dpF_total		-0.5349		
locM		-0.5643		-0.6257
loclM		-0.5975		-0.5321
ccM		-0.6454		
hbM		-0.6169		-0.5538
hdM		-0.6539		
heM		-0.6433		-0.5323
hlM		-0.6186		-0.5476
htM		-0.6433		-0.5323
hvocM		-0.5968		-0.5787
hvolM		-0.6169		-0.5538
nopM		-0.5936		
dpM		-0.6397		
mcM		-0.5958		
ecM			0.5075	

Figure 4.3: Correlation coefficients

necessarily require the developer to use functions. Sometimes the complexity analysis of files also failed.
Based on the average rank, the source files were put in order. For more convenience, the average rank is inverted to retrieve a testability score as a value between 0 and 100. A high value for this score means good testability. Finally, all analyzed source files in the dataset are put in a sequence ordered by the testability level. The list can be browsed in JSTT.
The best way to evaluate the scoring would be to write tests for some of the files, but this is too much additional work at this point. Without this, it is difficult to make accurate assumptions about the file's testability. Some of the files are randomly reviewed. What stands out is that files with a low score appear to be much longer than files with a high score. This result could be expected by the selection of metrics that are used. A smaller and less complicated file can be easier to test, but what also stands out is that sometimes the code is not accessible for other code outside of the file and thus is not well testable. Determining if a part of the code should be publically visible or not is not an easy task. Due to the many possibilities of exposing a method or variable, it is also very hard to determine the visibility programmatically. It can be declared with different syntax and even dynamically by adding or removing attributes from the public API during runtime. The testability score that is proposed here addresses code complexity as one aspect of testability but will need to be complemented with other characteristics that are hard to capture in metrics.
Using the testability scores, it is analyzed if certain kinds of projects are more testable than others. Therefore, the files are grouped by different attributes, which are the file types, the categories of the software that were determined earlier, the used test frameworks and more. Average file type and category ranks can be found in tables 4.1 and 4.2, and the other ones can be found in the source code of JSTT. What stands out regarding the file types is that TypeScript appears to be more testable than JavaScript. The *d.ts* files are ranked in the first place, but those files hold TypeScript declarations and not actual source code, so a lower complexity and thus a higher score is expected. The declarations can still be tested, even if that works differently from testing program code. TypeScript (TS) and React's TSX have a significantly higher average testability score than JavaScript (JS). React's JSX has a little lower score than TS and TSX but a significantly higher score than JS. The remaining three file types (mjs, gs, cjs) are only rarely used and thus do not rely on a big data source. While they can be considered to hold basic JavaScript code, they are ranked lower than TS. GS and CJS have very low scores and thus seem to have bad testability. The findings are an interesting addition to the work of Bogner and Merkel, who compared software quality of JavaScript and TypeScript using code smells, understandability and bug fixes. They found that TypeScript applications had higher code quality in terms of code smells and better understandability. [15] It seems like TypeScript developers are more skilled or at least put more effort into

Rank	File type	Average score	No. files	No. repos
1	d.ts	96.14	170	69
2	ts	90.43	2619	85
3	mjs	88.35	6	5
4	tsx	88.34	1202	37
5	jsx	86.76	345	13
6	js	77.28	7932	329
7	gs	69.19	15	2
8	cjs	52.24	10	3

Table 4.1: Average testability scores by file type

producing code of good quality. Maybe TypeScript is more often used by more experienced developers and beginners tend to stick with JavaScript. TypeScript is a superset of JavaScript and therefore has features that add complexity to the language. Those hypotheses will need to be validated in a separate study.
The average scores for categories shown in table 4.2 also reflect expectations concerning the criticality of the project. Projects categorized as crypto, shopping, security and payment have a significantly higher testability score than utility, creativity, browser extensions and social networks. Of course, this may only hold for the used sample of repositories with many being small and without much economic interest. With growing economic interest, the criticality may also increase for a certain project. Interestingly, algorithms appear to also have very low testability scores. The fact that the scoring system focuses on code complexity explains this circumstance because algorithms, e.g., for encryption, can be very complex.
For the average ranks of files per used test framework in the corresponding repository, *jest* as the most used framework is also in the top seven with an average rank of 89.10. The top five frameworks are only used in one or two repositories, so there might not be sufficient data to make repository-independent assumptions for them. On the other end, *jasmine* and *karma* have lower scores around 81. Files from projects with frontend code do not have a significantly different average score (79.50) than files from projects with backend code (81.82), backend projects perform slightly better.
It was also analyzed if the testability level of a file correlates with the star rating of the corresponding repository. A plot of the distribution of data can be found in appendix 1. For better readability, the distribution for the top-rated repository is shown in another figure in appendix 2. The correlation was computed using Spearman's RHO. With a correlation coefficient of -0.131 and a p-value of $2.81\text{e}-48$ there is no correlation found between the testability level and the star rating.
An interesting observation can be made when comparing the average scores of files in tested and untested projects. Files in projects that have at least one test case have a signif-

icantly higher average score (86.91) than files in projects without any test code (77.43). It follows that code generally is less complex and thus easier testable if developers have testing in mind.

Table 4.2: Average testability scores by category

Rank	**Category**	**Average score**	**No. files**	**No. repos**
1	Gastronomy	93.713	45	1
2	Parser	93.645	4	1
3	Crypto	92.596	1232	7
4	Security	88.813	1447	19
5	Payment	88.798	367	5
6	Shopping	88.29	210	5
7	Demonstration	88.207	1628	61
8	Science	86.628	26	3
9	Sports	86.07	7	1
10	Server	84.842	342	22
11	Documentation	84.603	148	12
12	Privacy	84.267	9	2
13	Real Estates	83.947	7	1
14	Development	83.461	2238	86
15	Blog	83.197	232	2
16	Network	83.163	63	4
17	Health	83.024	15	1
18	Data Transfer	82.882	214	25
19	Testing	82.552	215	11
20	Websites	81.304	4002	116
21	Database	81.206	155	6
22	AI	81.1	45	4
23	Mobile	80.821	101	4
24	Games	80.745	578	22
25	Multimedia	80.744	446	15
26	Plugin	80.499	571	23
27	Monorepo	80.41	1651	4
28	Graphics	80.254	5	1
29	Smart Home	79.961	226	3
30	Concurrency	79.862	3	1
31	Productivity	79.778	671	20
32	Utility	79.095	48	10
33	Creativity	75.204	253	13

Continued on next page

Table 4.2 – *Continued from previous page*

Rank	Category	Average score	No. files	No. repos
34	Trading	70.736	1	1
35	Browser Extension	69.455	123	8
36	Files	68.083	272	6
37	Social Networks	65.344	1096	12
38	Algorithms	64.182	61	6
39	Statistics	63.303	19	2

4.2.4 Comparison with Java

As described in chapter 2, existing work focused on analyzing testability in Java applications. Their findings are compared with the findings from this thesis for JavaScript. In the following, a metric is referred to with the name used in this thesis, the name used in the mentioned study is given in brackets if needed.
Alenezi found source code metrics that correlate with test case effectiveness represented by the mutation score. Looking at the intersection of metrics, for both languages, there is a strong correlation to the *locM*, *mcM* (NMC) and *ccF_total* (WMC) source code metrics. The strong correlation to *loccM* (LOCCOM) that Alenezi also found did not appear for JavaScript. [2] The work of Badri and Toure came to similar results. They calculated correlations between test and source code metrics and found *locM* (LOC) and *mcM* (NMC) to be significantly correlated to test metrics. [10]
Similar to the approach in this thesis, Bajeh et al. used test coverage metrics to find correlated source code metrics for object-oriented software. Interestingly, they observed only weak correlations and did not find metrics significantly related to testability, but they only used a small dataset with six Java applications and only included tests that achieved a code coverage of more than 60%. [11] Bruntink and van Deursen also checked the correlations of test metrics to source code metrics, using two Java applications as a data source. They found strong significant relations to *ecM* (FOUT), *locM* (LOCC), *noF* (NOM) and *ccF_total* (WMC). [18] Except for *noF* (NOM), the results were similar in this thesis. Filho et al. found a large list of strongly correlating metrics when studying the relation between test and source code metrics. This includes *locM* (LOC), *ecM* (EC), *acM* (AC), *ccF_total* (WMC), *noF* (NOM), *ccM* (CC) and *noM* (NOCL). The data is retrieved from two Java applications, from the reviewed literature it is the only work that found so many strong correlations. [30] For *acM* (AC), *noF* (NOM) and *noM* (NOCL), there was no strong correlation found for JavaScript.
The Halstead metrics that are used in this thesis were not used in the mentioned studies. Unfortunately, some studies rely on a small dataset of software projects. Apart from that, some metrics commonly appear in multiple studies as proposed indicators of testability,

namely *locM* (LOC), *ccF_total* (WMC), *ecM* (FOUT) and *mcM* (NMC). The same results arose for JavaScript. *noF* (NOM) and *noM* (NOCL) appeared multiple times in the Java studies but not for JavaScript, so there might be a difference originating from the scripting nature of JavaScript.

4.3 Refactorings for Testability

The dataset and the information described in the previous chapters form the basis to find refactorings for testability, aiming to answer the second research question. Assuming that developers need to make modifications to the source code to create, update or extend tests, the goal is to find those refactorings and analyze them regarding patterns and frequency.

4.3.1 Finding Commits

Repositories that were already found to not have tests are discarded for this task, leading to 145 repositories with 31006 commits and 12029 pull requests in total. The GitHub API allows filtering commits by a given filename. So for every pair of source and test files in a repository, it is searched for commits changing the source file and for commits changing the test file. Commits that are in the result set of both queries are saved for assessment. Additionally, for every one of those commits it is searched for pull requests that are associated with it. If existing, the pull requests are saved to give more information about changes in a commit if needed during the analysis.
A total of 802 commits that fulfill the criteria are found, with 340 associated pull requests. The average number of found commits per repository is approximately 5.53, with a median of 2 and a maximum of 61. For 26 repositories, no commits matching the criteria were found, leaving 119 repositories with commits.

4.3.2 Mining Refactorings

Unfortunately, for JavaScript there is very limited tool support for automated refactoring mining. Two tools were found that could be used.
The first one is a popular tool that has existed since 2017 and got an update in 2020 called RefDiff 2.0[4]. The authors also published some papers explaining the purpose and performance of the tool. [76] While the tool initially came with support for only Java, the current version also supports JavaScript and C. The authors state good precision and recall for JavaScript refactoring mining, but they did not find an annotated dataset for refactorings in commits for this language. Their design of the precision and recall

[4] `https://github.com/aserg-ufmg/RefDiff`

validation on very small and non-representative samples does not give enough proof for a good performance. The focus of the tool is still clearly Java. 10 different refactoring types are supported for JavaScript.
The second tool that can find refactorings in JavaScript commits is JsDiffer. It was developed as part of a master's thesis and is an adaption of a popular Java refactoring mining tool called RefactoringMiner 2.0[5]. The source code is not publically available but the author permitted the usage for this thesis. The tool has similar goals to RefDiff 2.0 but achieves lower performance in comparison. It supports 18 different types of refactorings.
The refactoring miners for JavaScript come with limitations and appear to stand behind their Java alternatives. One limitation was especially obstructive at first, which was the support of variations of the JavaScript language. The tools do not support TypeScript or React's variations of both forms. It was crucial to overcome this limitation and modify the tools to support TypeScript. The cores of both tools are written in Java, but both use JavaScript to parse JavaScript source code. Both make use of the Babel parser[6] which is one of the most popular parsers for JavaScript and has good support of the latest language features and a plugin architecture. Both tools were similarly modified so that they can additionally parse TypeScript and React's .jsx and .tsx files. Depending on the file's type, different plugins are loaded in the Babel parser and maximum file type support is achieved.
The second aspect that needed to be modified for both tools was the interface. The tools are Java projects and are expected to be used in Java applications. The tool used for nearly all tasks in this thesis - JSTT - is a NodeJS application and should be able to communicate with the refactoring miners. So both miners are wrapped with a minimal REST API using the Spring Boot framework[7], exposing one endpoint that expects the name of a repository and a list of commit identifiers (SHAs) and returns the identified refactorings for each commit. Due to compatibility reasons, the RefDiff API was set up as an IntelliJ project and the JsDiffer API was set up as an Eclipse project.
For all test-pair commits that were determined previously, refactorings were mined with both tools. The mining process is automated in JSTT and the results are stored per repository and commit in a JSON file. Error scenarios were evaluated and the tools could be adapted in some cases to be able to process most of the files in commits. The adaptations were mostly configurations of the used plugins in the Babel parser. RefDiff appeared to be the more robust tool meaning it produced fewer processing and runtime errors. JsDiffer sometimes froze without warnings while processing specific files, so those needed to be skipped manually. The presence of quite some error scenarios in combination with the lower maturity of the miners indicates the need for manual analysis

[5] `https://github.com/tsantalis/RefactoringMiner`
[6] `https://babeljs.io/docs/babel-parser`
[7] `https://spring.io/projects/spring-boot`

in addition to automated refactoring mining. Errors mostly appeared while parsing and the reasons were often illegal or deprecated syntax or encoding.
The refactoring miners found a total of 3379 refactorings distributed around 252 commits in 67 repositories, with an average of 23.30 refactorings per repo and 12.75 per commit. Table 4.3 shows insights into the found refactorings. It shows the total number of refactorings per type, which tool found them and how many commits contained refactorings of the type. JsDiffer supports more refactoring types and found about 85% of the refactorings outputted by both tools, but many of them have no relevance to this work. Refactorings that just rename anything will not enable writing tests. They may lead to better understandability which may make testing easier for a developer, but as already stated, this work will only consider refactorings to be for testability if they were clearly done for this purpose. Especially for automated test generation, the names do not matter. Moving a file is also not interesting here because it does not matter where a file is located to test it. The same holds for most other refactorings that move methods and classes, if the move is done within the same block level. The remaining refactorings are the most promising ones to find refactorings for testability.

Table 4.3: Mined refactorings

Total	**Type**	**Tool**	**No. commits**
1059	Move File	JsDiffer	31
528	Rename Variable	JsDiffer	74
377	Move Class	JsDiffer	12
254	Rename File	JsDiffer	32
148	Rename Method	JsDiffer	41
146	INTERNAL_MOVE	RefDiff 2.0	18
123	Rename Parameter	JsDiffer	24
122	RENAME	RefDiff 2.0	54
122	MOVE	RefDiff 2.0	29
94	Add Parameter	JsDiffer	38
94	Remove Parameter	JsDiffer	16
69	Move And Rename File	JsDiffer	10
68	Move Method	JsDiffer	6
48	EXTRACT	RefDiff 2.0	25
32	MOVE_RENAME	RefDiff 2.0	17
29	Move And Rename Class	JsDiffer	10
18	EXTRACT_MOVE	RefDiff 2.0	8
17	Extract Method	JsDiffer	11
10	Rename Class	JsDiffer	5
10	Inline Method	JsDiffer	2

Continued on next page

Table 4.3 – *Continued from previous page*

Total	Type	Tool	No. commits
7	INLINE	RefDiff 2.0	5
2	Change Variable Kind	JsDiffer	2
1	INTERNAL_MOVE_RENAME	RefDiff 2.0	1
1	Extract And Move Method	JsDiffer	1

4.3.3 Commit Analysis

To enable a systematic and structured analysis of commits, JSTT is extended to provide a UI to work with commits and refactorings (shown in appendix 6 and 7). An overview of all commits allows for sequentially classifying all commits. Commits can be marked as done or just for attention at a later point in time. The progress and the found refactorings for testability are tracked. Commits can be opened in a detail page, where all refactorings, a link to the GitHub page and associated pull requests are shown. Refactorings can be commented on and marked as refactorings for testability. Additional refactorings that were not found by the mining tools can be easily added. The comprehensive UI gives an intuitive representation of all relevant data and provides a straightforward workflow for classification.
In the first step, all 252 commits that had mined refactorings were manually reviewed and analyzed. It was checked if any of the mined refactorings were done for testability. Commit messages as well as connected pull requests and their descriptions helped to understand what was done. The approach for reviewing depended on some aspects of the commit. Source code files that were added in the commit can not contain refactorings, if the code introduces new features and the file creation was not due to code extraction from an existing module. Often, it was helpful to first look into the test files and search for changes that could require changes in the source code. If there were huge modifications in the test files, it was helpful to first look into the corresponding source file and search for refactorings. In addition to the review of mined refactorings, it was searched for other refactorings that were done for testability.
After that, the remaining 550 commits for which the mining tools could not find any refactorings were analyzed. Code changes were reviewed in the same way as for the first 252 commits and it was manually searched for refactorings for testability. In some cases, it was easy to skip the commit because files were added but not modified, especially in initial commits. In other cases, it took a long time to understand what was done or to review all of the changes in large commits. Examples of changes where it was hard to decide if they are testability refactorings are methods that were added to the public interface to expose some values or other internals that already existed before the commit. Especially when those new methods are not used in the source code but in the tests, it was a hard decision if the method is a new feature or if it is needed for testing. Some

Total	Type	No. commits	No. repos
11	Add parameter	6	5
10	Internal move	1	1
5	Export function	5	4
3	Extract	3	3
3	Set test environment	1	1
2	Move rename	1	1
2	Add getter	2	2
1	Wrap	1	1
1	Add return value	1	1
1	Add attribute	1	1
1	Export object	1	1
1	Move config to object	1	1
1	Split module	1	1

Table 4.4: Testability refactorings

repositories consist of libraries that can be used by other applications, so new methods can be just new features for the API. In those cases, a deeper understanding of the source code and the commit context was essential.
Some observations could be made when reviewing the test files. Tests were mostly unit tests, some were integration tests. Performance tests were rare and security tests could not be found. The use of assertions seems to be uncommon as well. Assertions are not a native concept in the ES language specification, so external libraries need to be utilized when assertions should be used.
In total, 42 refactorings for testability in 22 different commits from 16 different repositories were identified. They are shown in table 4.4 and a complete list of the refactorings for testability with their locations can be found in appendix table 1. That makes about 2.7% of commits that include testability refactorings. A big difference regarding the number of results exists between the commits with refactorings identified by the mining tools and the commits without those refactorings. From the 252 commits with mined refactorings, 13 commits (approximately 5.2%) contained a total of 30 testability refactorings. Accordingly, 9 (approximately 1.6%) of the remaining 550 commits contained a total of 12 testability refactorings. 27 of the testability refactorings were not found by the mining tools but instead within the manual reviews. That shows the need for manual reviews instead of relying on the results from the tools. It confirms the previously stated observation that the tooling for JavaScript in this field is underdeveloped. The tools need to support more refactoring types and have better accuracy.
To improve the reliability of the achieved results, 71 of the 802 commits were additionally reviewed by the supervisor of this thesis. The selection of commits was partly

random, while it was assured that some commits did contain testability refactorings regarding the results of the author of this thesis. He was asked to review them to compare the results with those of the author of this thesis. The first comparison took place after the supervisor reviewed the first half of the 71 commits. When different results for a commit appeared, they were reevaluated to achieve consensus. The same holds for commits where the decision was not clear. After the comparison, the supervisor reviewed the second half of the commits. The results were compared again. The second reviewer did find one additional refactoring for testability but missed some of the refactorings identified by the author of this thesis. Every commit that was found to contain refactorings for testability was reviewed by the author of this thesis at least three times on different days.

The low number of testability refactorings is plausible because the used approach does not focus on commits that are likely to introduce refactorings or testability improvements. The approach aims to get complete results for the selected sample. That makes it more difficult to compare the findings with the work by Reich and Maalej [71], who used keywords to search for potentially relevant pull requests. In their sample, they found that approximately 25% of the analyzed pull requests contained testability refactorings, which is significantly more. Apart from the pull requests selected by keywords, they also analyzed a small random sample of pull requests that modify test and source code files simultaneously. They found that 12.7% of the random pull requests included refactorings for testability. This number is still significantly higher than the 2.7% of commits containing testability refactorings identified in this thesis. This opens the question of the reasons for the difference. Possible reasons may be that in JavaScript projects, fewer refactorings for testability are done, that the used datasets differ in their creation process or that pull requests contain refactorings for testability more frequently.

To check if a keyword-based approach would have worked for the sample in this thesis, it was searched for the same keyword combinations associated with testability in commits and pull requests. The commits and pull requests are the same which led to the results described before. In commits, the commit message is used and in pull requests, the title and the body are used. The search was done case-insensitively. Searching for the word *testable* led to zero results, the same holds for the word *testability*. Searching for commits whose messages contain both keywords *test* and *refactor* gave 14 results. Only one of those commits contains exactly one refactoring for testability. Searching for the same keyword combination in pull requests gave 29 matches, but those did not contain testability refactorings. It is proved that the keyword-based approach does miss most of the refactorings for testability, which is an important disadvantage. Developers seem not to mention this special type of refactoring when describing their changes. Another problem is that a certain amount of repositories are using other languages than English, so English keywords will not fit at all. The accuracy of the keyword search was very low. It remains unclear, if refactorings for testability are less frequent in JavaScript

projects than in Java projects or if developers miss out on mentioning them. To answer this question, the same approach that Reich and Maalej used would need to be studied on a larger scale for JavaScript.
As table 4.4 shows, four refactoring types occurred in more than one commit. The *Internal Move* refactoring has ten occurrences but all within the same commit. The *Add Parameter* refactoring appears to be the most frequently used one in the analyzed sample. The *Export Function* refactoring is used when an existing or new function is added to the public interface and occurs five times. The *Extract* refactoring is used when code is extracted from a scoped block to exist on its own. This often means that code is extracted from a function and put into its own function. Three times this refactoring was done for testability. All the testability refactorings are described in more detail in the following when refactoring patterns are extracted from the findings. What stands out is that only two refactorings improve UI testability.

Name	**Test type**	**No. TRs**	**Associated refactoring types**
Add parameter	Unit	11	Add parameter
Wrap code	Unit	11	Internal Move, Wrap
Widen access	Unit	3	Export function, Export object
Create helper	Unit	3	Export function
Use environment variables	Unit	3	Set test environment
Extract code	Unit	6	Move rename, Extract, Split module
Add getter	Unit	2	Add getter
Add return value	UI	1	Add return value
Programmatic configuration	Unit	1	Move config to object
Add property to identify UI element	UI	1	Add attribute

Table 4.5: Refactoring procedures to improve testability

4.3.4 Testability Refactoring Procedures

To use the findings to help developers improve the testability of JavaScript code, procedures that apply in certain scenarios are extracted from the refactorings. They will not be called patterns because the term pattern is often used for something that can be observed frequently. Unfortunately, the procedures rely on a small source of data, so it is not clear if they are likely to be commonly used across all JavaScript projects. If necessary, the

procedures are illustrated using real-world examples. Only the relevant parts of the real-world code are shown, so the examples are not complete. The procedures are extracted from the testability refactorings that were identified under the assumption that they are useful. Table 4.5 summarizes the procedures and the number of Testability Refactorings (TRs) that form the foundation of the procedures. The table also shows for which types of tests the procedures were used. The types of tests stated there do not necessarily mean that the procedures do not apply to other tests, but there was no evidence found.

Add parameter

The most frequent refactoring for testability can either consist of a single or a composite refactoring. Adding a parameter to a function or class constructor allows passing additional data. Depending on the context, the parameter may be added to other functions as well, making the procedure a composite refactoring consisting of multiple *Add Parameter* refactorings. A reason to use this procedure can be that a mocked object should be used in the function during testing. Another reason could be that any default value should be overwritten during testing. When the parameter is not necessarily needed in the production environment because it would always hold the same value, a parameter default can be used. In the following, a reduced example of a parameter that is added to a constructor to allow the use of a mocked function is shown.[8] Figure 4.1 shows the constructor before refactoring. In figure 4.2, a parameter for a fetch function is added to allow overwriting the default fetch function. In the associated test file shown in figure 4.3, a simple mock function is passed to prevent the default behavior.

```
1    constructor(jwtToken: string) {
2      this._jwtToken = jwtToken;
3    }
4
```

Figure 4.1: BffApiClient.js (before)

```
1    constructor(jwtToken, fetchFunc = null) {
2      this._jwtToken = jwtToken;
3      this._fetch = fetchFunc || fetch;
4    }
5
```

Figure 4.2: BffApiClient.js (after)

[8] https://github.com/ably-labs/fully-featured-scalable-chat-app/commit/1dd9eab2b88591789ec2bbf869178b1c5b030fd1

```
beforeEach(() => {
  fetchMock = () => { return fetchResponse; };
  sut = new BffApiClient("token", fetchMock);
});

```

Figure 4.3: BffApiClient.test.js

Wrap code

This procedure has similar motivations to *Add Parameter*. It consists of two steps. At first, a wrapper which could be a function or a class is created. It should have at least one parameter to pass some value that can be used in the class or function body. The second step is to move existing code into the created wrapper to allow the use of the parameter value. The code may need to be modified so that it makes use of the parameter value. The procedure can be used when the creation of an execution context for some code is needed. This procedure is an extension of *Add Parameter* where the signature to which the parameter is added needs to be created first. In the example below, the procedure is illustrated in a reduced version of real-world usage.[9] Figure 4.4 shows a function before refactoring. In figure 4.5, the function is wrapped in another function that has a version parameter. The version is used to give different outputs in the wrapped function. While this can be used to use different versions in the production environment, this can also be used to provide different behavior in a test context. Figure 4.6 shows that the wrapper is called with a test version parameter. The example is just used to demonstrate the idea, the real usage is not as simple as illustrated here and includes multiple atomic *Internal Move* and *Add Parameter* refactorings.

```
function getBaseEnchantmentLevel( itemId, materialId ) { ... }

```

Figure 4.4: mc.js (before)

[9] https://github.com/Zazcallabah/mce/commit/474cb2e1aac9dfcde7af02f8556828ad090e5a20

```
var makeMC = function(version) {
  function getBaseEnchantmentLevel( itemId, materialId ) {
    if( version === "test" ) {
      return ...;
    }
    if( version === "1.2" ) {
      return ...;
    }
    if( version === "1.3" ) {
      return ...;
    }
  }
}

```

Figure 4.5: mc.js (after)

```
var _mc = makeMC("test");

```

Figure 4.6: test_simulate.js

Widen access

This procedure consists of one atomic refactoring that addresses problems with the visibility of functions. It can be used when a class or function needs to be tested on its own but is not accessible by the test code. It can be also used for other private variables and objects. In the following, it is referred to all these types as objects, because in JavaScript, functions and classes are also objects under the hood. Making objects public can be achieved in different ways. First, it should be noted that importing and exporting modules in JavaScript works differently in ES6 or CommonJS modules. In modern ES6 modules, the keywords *import* and *export* can be used. In CommonJS, importing needs to be done with the *require* function and exports need to be added to *module.exports*. So making an object accessible from outside the module can be achieved by using those language features and exporting it. If the object to be exported is a private member of a class, it can easily be converted to a public member. While in TypeScript there are the reserved keywords *public*, *private* and *protected*, in vanilla JavaScript this is achieved with a little syntactic difference. Class members are public by default but can be made private by prepending a # to the name. To make any other value public it can be added to the *this* object which refers to the class instance.

Create helper

Sometimes it is useful to create a helper function in the source code that does some task which is especially useful in testing. The helper has full access to private properties inside the source code, so that can be necessary or at least an easier way than doing complicated things in the test code to achieve the same result. Of course, the helper needs to be accessible from outside the module. A common example of a helper is a reset function, as it is added in the example[10] in figure 4.7. Like it is shown in figure 4.8, tests can use the same instance of *stats* this way without making changes to it that influences other tests.

```
stats._reset = function(){
  store = {};
  pipes = 0;
  stats.enabled = true;
};

```

Figure 4.7: stream/stats.js (after)

```
test('logging starts on pipe', function(t) {
  stats._reset();
  ...
});

```

Figure 4.8: test/stream/stats.js

Use environment variables

Telling the code that it should execute in a test environment can be achieved in many ways. With *Add Parameter* and *Wrap Code*, two of them are already explained. Another possibility is to use environment variables. This allows to set a variable that is available in the whole application without modifying interfaces. Figure 4.9 shows an example[11] where the environment variable is checked to determine which code to execute.

[10] https://github.com/pelias/openstreetmap/commit/571b6d2009b37bd2e01f3f0f78b29a2f6dde6bef

[11] https://github.com/BreakOutEvent/breakout-frontend/commit/1b78b5829628d86a900f7270d7e617e309e69252

```
global.IS_TEST = process.env.FRONTEND_RUN_TESTS === 'true';
if (!IS_TEST) {
  ...
} else {
  ...
}

```

Figure 4.9: app.js (after)

Extract code

In application design, there are patterns like Model View Controller (MVC) and Model View ViewModel (MVVM) that are supposed to separate code by responsibilities. Also for distinct features, it makes sense to separate their code. Separation not only makes sense at the module scope but also function scope. Tests should also test distinct functionalities, sometimes leading to the need for refactorings to improve cohesion or enforce design patterns. It may also be necessary to split functions or extract parts of them into a new one. In testing, this can have multiple benefits. The first advantage is that each part can be tested independently. That may improve test effectiveness, simplify tests and increase test coverage. The second advantage is that it is possible to replace a certain part of the logic with a mock. An example of the mocking can be seen below.[12] Figure 4.10 shows the function that is created from existing code and added to the public class interface. Figure 4.11 shows the test code where this function is replaced with a mocked version.

```
makeQuestion(question) {
  ...
}

```

Figure 4.10: gravity.js

```
gravity.makeQuestion = (question) => {
  ...
};

```

Figure 4.11: gravity.spec.js

[12] https://github.com/jupiter-project/gravity/commit/9620c4a94899d7d7cb416b071d2a5462f7ea84fd

Add getter

This procedure refers to the addition of a specific type of function, which is the getter function. Getters are used to allow retrieving the values of private and inaccessible variables. There are different reasons why a getter is needed in tests. The first one is that the test should check if a certain value is correct during or after the execution of some code. The second reason is that the getter function should be mocked to return a different value in tests.

Add return value

Increasing the observability of code, this procedure applies when functions have no return value. When a function is called in tests, it is common that the outcome should be checked. There are multiple ways to inspect the outcomes which heavily depend on the context. But there are cases where it is beneficial if the function returns data that allows checking out the outcome directly.

Programmatic configuration

It is a common practice for some libraries to load configuration options from a file. In testing, it may be cleaner if the creation of such a file is not needed and the configuration can be done programmatically in the test code. In the following, an example[13] is illustrated. Figure 4.12 shows a function that expects the name of a config file and loads this file from the given directory. Figure 4.13 shows the modified version that expects the config as an object. This way the complicated way of creating a configuration object and saving it to the file system shown in figure 4.14 is not needed anymore. Now, the configuration can just be passed to the function as an object like it is illustrated in figure 4.15.

```
export const buildBook = async (
  configFilename: string,
  projectDir = path.dirname(configFilename),
) => {
  const config: ConfigJson = await readConfig(configFilename)
  ...
}

```

Figure 4.12: build-book/index.ts (before)

[13] https://github.com/erukiti/easybooks/commit/992eff78e886533b74cbce3e3eeaf9af5301da5d

```
export const buildBook = async (config: ConfigJson, projectDir:
string) => {
  ...
}
```

Figure 4.13: build-book/index.ts (after)

```
const conf = JSON.stringify({ ... })
writeFile(path.join(tmpDir, 'test.json'), conf, {
  encoding: 'utf-8',
})
await buildBook(path.join(tmpDir, 'test.json'))
```

Figure 4.14: build-book.test-harness.ts (before)

```
const conf: ConfigJson = { ... }
await buildBook(conf, tmpDir)
```

Figure 4.15: build-book.test-harness.ts (after)

Add property to identify UI element

One form of a basic UI test checks if the UI renders correctly. To test if a UI-component renders correctly, it needs to be possible to identify it in the DOM. That might be difficult if the output for the component is just a structure of basic HTML elements. Instead of validating a complex structure, the component could just receive an attribute that identifies it. Figures 4.16 and 4.17 show how the *role* attribute is added to the component, figure 4.18 shows the test that can now find it using the attribute.[14]

```
const LoadingContainer = styled.div.attrs(({ className, light })
=> ({
  className: ...
})
```

Figure 4.16: Loader.js (before)

[14] https://github.com/smswithoutborders/SMSWithoutBorders.com/commit/e1ebdfbbd9eb5e5b7e3b353cbf8637ab54c01b45

```
const LoadingContainer = styled.div.attrs(({ className, light })
=> ({
  role: 'alert',
  className: ...
})
```

Figure 4.17: Loader.js (after)

```
it("Renders Loader", async () => {
  render(<Loader message="processing ..." />);
  expect(screen.getByRole("alert")).toBeInTheDocument();
  ...
});
```

Figure 4.18: Loader.spec.js (after)

4.3.5 Comparison with Java Patterns

As already mentioned, there is one similar study on refactorings for testability by Reich and Maalej. [71] They identified ten refactoring patterns in the analyzed pull requests. In the following, similarities and differences to the procedures proposed in this thesis are discussed. Proposals from this thesis are called procedures and their proposals are called patterns.
Starting with the similarities, the *Widen access* procedure can be compared to the *widen_access_for_invocation* and *widen_access_for_override* pattern, all three address visibility. The patterns are separated by their purpose, which is either override or invocation. In this thesis, the different purposes were combined into one procedure due to the small number of related refactorings for testability.
The *Extract code* procedure combines four patterns, which differ in their target of extraction and again in their purpose. *extract_class_for_override* and *extract_method_for_override* are used to be able to mock the class or method. *extract_class_for_invocation* and *extract_method_for_invocation* are used when a class or method is not visible to the test code.
The *add_constructor_param* pattern is used to create a class instance with additional dependencies. The *Add parameter* procedure is similar but not limited to constructors. In JavaScript, constructors have different importance than in Java because of the different language concepts that are already described in detail in chapter 2.
The *Add getter* procedure has similarities to the *extract_attribute_for_assertion* pattern. For both, the goal is to give access to an internal variable by adding a getter function.
Some procedures or patterns do not occur in both studies. In JavaScript, it is not pos-

sible to overload functions or constructors. In Java, overloading is allowed. The *create_constructor* pattern, as a variation of the *add_constructor_param*, is used to create another constructor for testing. A similar procedure for JavaScript will not apply due to the mentioned restrictions.
A procedure that appears to be relevant in JavaScript but not in Java is *Wrap code*. In Java, code is always wrapped in classes, but JavaScript is a scripting language and may require to create wrappers for certain parts of the code.
Only two procedures for UI testing were found, which are *Add return value* and *Add property to identify UI element*. There is no sufficient data to compare JavaScript and Java regarding procedures and patterns for UI testing. JavaScript was invented for browser usage and therefore UIs play an important role in frontend JavaScript code, but Java can also be used to create UIs.
The remaining procedures and patterns are expected not to be language-specific but were not found or maybe not searched for in the respective other analyses. For example, the *Create helper* procedure is used to improve controllability and did not occur in the Java study, but it might apply to Java as well. When comparing the findings from both studies, it stands out that the patterns and procedures often solve similar testability problems. They are supplemented by patterns and procedures that evolve due to the particularities of the languages.

5 Conclusion

In this thesis, two research questions were answered using a newly created dataset of GitHub repositories. The dataset is built using random sampling to fulfill the need for a representative sample of open-source JavaScript repositories on GitHub with five or more stars. It is annotated with meta information like categories, existence of tests, dependencies, test frameworks and more. 39% of the repositories in the dataset contain at least one test case. A dataset like this appeared to be rare during the literature review, so the dataset can be further used in future studies.
The first research question addresses the level of testability in JavaScript open-source projects and the characteristics of testable JavaScript. Using correlation analysis of test and source code metrics, a set of testability-related metrics was established. The selection of metrics appeared to be similar to existing studies done for Java applications. Those metrics can be monitored by developers to help create more testable source code. A scoring system was proposed to allow assessing the level of testability of source code files. Using this scoring system, the source code files in the dataset for which metrics could be calculated were evaluated and compared by different attributes. Outstanding findings are that TypeScript code has better testability than JavaScript code on average. The same holds for projects with higher requirements for the absence of bugs in categories like crypto, shopping, security and payment compared to categories like utility, creativity, browser extensions and social networks. Additionally, projects that contain test cases appear to have better testable code than projects without tests. The drawback of the scoring system is that testability is a complex quality attribute and the scoring system does not address all aspects. It focuses on code complexity as part of testability but misses aspects like the visibility of methods. In summary, the approach allows the comparison to similar work but it is still no complete solution for testability assessment.
Using the same dataset, refactorings for testability were studied to answer the second research question. The goal was to check how developers improve their code to achieve better testability. An approach that aims not to miss those refactorings is followed, leading to good but few results. Commits that modify related source code and test files were analyzed with the help of associated pull requests. The approach was compared to another existing approach that utilizes keyword search to find pull requests containing refactorings for testability in a bigger dataset. While the other approach may achieve a higher precision when applied to a big dataset, it was proven that it would have missed most of the refactorings for testability in the sample used in this thesis.
The analysis of the commits showed that it is quite difficult to find properties that define commits with testability refactorings, but some insights would help if the study was

repeated. A main improvement would be reducing the manual effort. A part of this is a higher precision of the commit selection without worsening the recall. That would allow the inclusion of more repositories and therefore commits in the initial dataset without increasing the manual effort. Some types of commits like initial commits and commits that do not change but only create source code files of interest can be ignored. Another part is the refactoring detection. Some refactoring types appeared not to be used for testability improvements, those can be ignored. An improved refactoring detection would also bring huge benefits and could decrease manual effort drastically. It came out that the majority of refactorings for testability were not supported by the mining tools. If they were supported, there would be no need for manually reviewing all code changes in commits.
Extracted from the findings, ten refactoring procedures were proposed that developers can use to improve the testability of their code. Since those procedures rely on a relatively small dataset of refactorings for testability with sometimes only one underlying occurrence in the analyzed commits, more code will need to be analyzed to verify their relevance. That could be achieved with better commit selection and refactoring detection as described above.
In addition to the mentioned contributions to this field of study, this thesis ships with an extensive and comprehensive tool (JSTT[1]) that can visualize much of the examined data and automates many common tasks that are also interesting in other research. The creation of the dataset is easily replicable not only for JavaScript but for any language that is present on GitHub. This way, random datasets of varying sizes and restrictions can be created with very little effort. This may support JavaScript testability gaining more attention in academic research due to the lack of similar datasets. Aside from the dataset creation, the tool has an extensive REST-API for tasks like metric collection, commit and pull request fetching and querying the GitHub API and NPM registry. Some exemplary screenshots of JSTTs UI can be found in the appendix.
Derived from JSTT, a little command-line tool called JavaScript Testability Score (JSTS) is built that analyzes JavaScript and TypeScript projects and calculates testability scores for the corresponding files. It uses the ranking data that is collected from the dataset described in section 4.2.3 as a basis. It can be used by a developer to assess the testability of his code, also calculated metrics and their scores can be outputted. The usage is documented and JSTS is publically available in the NPM registry[2].

5.1 Threats to Validity

Due to the creation procedure of the dataset used in this thesis, it is considered to be representative of the mentioned sampling frame. The drawback of including less pop-

[1] `https://github.com/Caramba997/JSTT`
[2] `https://www.npmjs.com/package/js-testability-score`

ular repositories than most other studies may be lower code quality. In the correlation analysis and also the search for testability refactorings it may negatively influence the results when source code and tests are not following standards of good programming. It is difficult to find characteristics of testable code when the tests used in the analysis are missing test cases or quality because the developer did not aim to achieve good test coverage and quality. In more popular repositories that will be less of a problem because the code is reviewed and maintained by more people and the interest in good quality is higher.
In the course of this thesis, a lot of tasks were automated using custom scripts. Even though they were developed while aiming for correctness and the results were reviewed carefully, the absence of errors can not be guaranteed. Errors in the scripts may influence the outcomes and thus the findings of this thesis.
The correlation analysis as well as the search for commits are based on manually established connections of source code and test files. For unit tests, this was often not a big problem, but there were test files that could not be assigned to a source code file that easily. It is also not always possible to do a one-to-one mapping. The final connections may contain some errors or miss some cases. That might slightly influence the calculated correlations or lead to missing refactorings for testability.
The review of commits to find testability refactorings was mainly done by one person, which is the author of this thesis. More than three years of professional experience in client- and server-side JavaScript programming are good preparation for the task but may still be insufficient to understand all the different facets of the language that are used across the repositories. To minimize the risk of falsely labeling commits, the supervisor of this thesis also analyzed a smaller sample of the commits. While he has great experience in programming and also studies refactorings for testability, he is not that familiar with JavaScript. Those circumstances may lead to missed occurrences in the examined code. The chance of falsely classifying refactorings as refactorings for testability is expected to be low because only occurrences that clearly address testability in the opinion of the author of this thesis are labeled as such and are evaluated multiple times.

5.2 Future Work

While testability is extensively studied for the Java language, this thesis continues exploring the research topic for JavaScript. There are many possibilities to kick off new studies from here. One thing that should be done is to replicate more Java studies to further analyze the similarities and differences of testability in both languages. It would also be interesting to use other sources to create a dataset like SourceForge or NPM to see if this leads to different results. Regarding testability metrics, it remains unclear if the outcome will be the same if a dataset with more popular repositories is used because

they are expected to have code with higher quality. The proposed scoring system mainly focuses on source code complexity, but other aspects like the accessibility of functions or variables from outside of the module are important too.
The findings from this thesis indicate some differences in testing and testability between JavaScript and TypeScript, which were also recognized in other studies. TypeScript appeared to be more testable. Future work could dive deeper into the reasons for those differences, e.g., if the skill set of the developer is a relevant factor.
Since only a few refactorings for testability were found in this thesis and the research in this field is generally sparse, more work should be done on this topic. Furthermore, the proposed refactoring procedures should be verified using a bigger data source and tested by real developers. Once a set of relevant patterns is created and verified, a tool that can be practically used by JavaScript developers to guide them in writing testable code can be implemented. That could consist of real-time monitoring of an overall testability score and identified problems and additionally proposed refactorings to improve existing code.

References

[1] Feras Al Kassar et al. "Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications". In: *Network and Distributed System Security (NDSS) Symposium* (2022). URL: https://www.ndss-symposium.org/wp-content/uploads/2022-150-paper.pdf (visited on 03/15/2023).

[2] Mamdouh Alenezi. *Investigating Software Testability and Test cases Effectiveness*. 2022. DOI: 10.48550/ARXIV.2201.10090.

[3] Mohammad AlMarzouq, Abdullatif AlZaidan, and Jehad AlDallal. "Mining GitHub for research and education: challenges and opportunities". In: *International Journal of Web Information Systems* 16.4 (2020), pp. 451–473. ISSN: 1744-0084. DOI: 10.1108/IJWIS-03-2020-0016.

[4] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. "Can Refactoring Be Self-Affirmed? An Exploratory Study on How Developers Document Their Refactoring Activities in Commit Messages". In: *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. 2019, pp. 51–58. DOI: 10.1109/IWoR.2019.00017.

[5] Nadia Alshahwan et al. "Improving Web Application Testing using testability measures". In: *2009 11th IEEE International Symposium on Web Systems Evolution*. Sep., 2009, pp. 49–58. DOI: 10.1109/WSE.2009.5630393.

[6] Paul Ammann and Jeff Offutt. *Introduction to software testing*. 2nd ed. Cambridge University Press, 2016. ISBN: 1316773124, 9781316773123.

[7] Esben Andreasen et al. "A Survey of Dynamic Analysis and Test Generation for JavaScript". In: *ACM Comput. Surv.* 50.5 (2017). ISSN: 0360-0300. DOI: 10.1145/3106739.

[8] Andrea Arcuri and Juan P. Galeotti. "Enhancing Search-Based Testing with Testability Transformations for Existing APIs". In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (2021). sep. ISSN: 1049-331X. DOI: 10.1145/3477271.

[9] Linda Badri, Mourad Badri, and Fadel Toure. "An empirical analysis of lack of cohesion metrics for predicting testability of classes". In: *International Journal of Software Engineering and Its Applications* 5.2 (2011), pp. 69–85. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b621e9c7e8fe26e2e355505ffb29cdaf14aa358a (visited on 03/15/2023).

[10] Mourad Badri and Fadel Toure. "Empirical Analysis of Object-Oriented Design Metrics for Predicting Unit Testing Effort of Classes". In: *Journal of Software Engineering and Applications* Vol.05No.07 (2012). 19738, p. 14. DOI: 10.4236/jsea.2012.57060.

[11] AMOS O. Bajeh et al. “Object-oriented measures as testability indicators: An empirical study”. In: *J. Eng. Sci. Technol* 15 (2020), pp. 1092–1108. URL: https://www.researchgate.net/profile/Abdullateef-Balogun/publication/340580611_OBJECT-ORIENTED_MEASURES_AS_TESTABILITY_INDICATORS_AN_EMPIRICAL_STUDY/links/5e91f889299bf13079913c9d/OBJECT-ORIENTED-MEASURES-AS-TESTABILITY-INDICATORS-AN-EMPIRICAL-STUDY.pdf (visited on 09/02/2023).

[12] Sebastian Baltes and Paul Ralph. “Sampling in software engineering research: a critical review and guidelines”. In: *Empirical Software Engineering* 27.4 (2022), p. 94. ISSN: 1573-7616. DOI: 10.1007/s10664-021-10072-8.

[13] Robert V. Binder. “Design for testability in object-oriented systems”. In: *Communications of the ACM* 37.9 (1994), pp. 87–101. URL: https://dl.acm.org/doi/pdf/10.1145/182987.184077 (visited on 03/14/2023).

[14] Jürgen Bitzer, Wolfram Schrettl, and Philipp J.H. Schröder. “Intrinsic motivation in open source software development”. In: *Journal of Comparative Economics* 35.1 (2007), pp. 160–169. ISSN: 0147-5967. DOI: 10.1016/j.jce.2006.10.001.

[15] Justus Bogner and Manuel Merkel. “To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and Typescript Applications on GitHub”. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 658–669. ISBN: 9781450393034. DOI: 10.1145/3524842.3528454.

[16] Hudson Borges, Andre Hora, and Marco Tulio Valente. “Understanding the Factors That Impact the Popularity of GitHub Repositories”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Oct, 2016, pp. 334–344. DOI: 10.1109/ICSME.2016.31.

[17] Tiago Brito et al. *Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages*. 2023. DOI: 10.48550/ARXIV.2301.05097.

[18] M. Bruntink and A. van Deursen. “Predicting class testability using object-oriented metrics”. In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. Sep., 2004, pp. 136–145. DOI: 10.1109/SCAM.2004.16.

[19] Mel Ó. Cinnéide, Dermot Boyle, and Iman Hemati Moghadam. “Automated Refactoring for Testability”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. March, 2011, pp. 437–443. DOI: 10.1109/ICSTW.2011.23.

[20] Flávia Coelho et al. “An Empirical Study on Refactoring-Inducing Pull Requests”. In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’21.

New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450386654. DOI: 10.1145/3475716.3475785.

[21] Mark E. Daggett, ed. *Expert JavaScript*. Berkeley, CA: Apress, 2013. ISBN: 978-1-4302-6098-1.

[22] Chetan Desai, David Janzen, and Kyle Savage. “A Survey of Evidence for Test-Driven Development in Academia”. In: *SIGCSE Bull* 40.2 (2008). jun, pp. 97–101. ISSN: 0097-8418. DOI: 10.1145/1383602.1383644.

[23] Wei Ding et al. “How Do Open Source Communities Document Software Architecture: An Exploratory Survey”. In: *2014 19th International Conference on Engineering of Complex Computer Systems*. Aug, 2014, pp. 136–145. DOI: 10.1109/ICECCS.2014.26.

[24] Jon Edvardsson. “A survey on automatic test data generation”. In: *Proceedings of the 2nd Conference on Computer Science and Engineering*. 1999, pp. 21–28. URL: https://faculty.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Readings/test.data.genration.survey.pdf (visited on 09/02/2023).

[25] Mahmoud Efatmaneshnik and Michael Ryan. “A Study of the Relationship between System Testability and Modularity”. In: *INCOSE International Symposium* 26.1 (2016), pp. 1922–1931. DOI: 10.1002/j.2334-5837.2016.00270.x.

[26] Gerald D. Everett and Raymond McLeod Jr. “Software testing”. In: *Testing Across the Entire* (2007).

[27] Amin Milani Fard and Ali Mesbah. “JavaScript: The (Un)Covered Parts”. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2017, pp. 230–240. DOI: 10.1109/ICST.2017.28.

[28] Michael Felderer and Ina Schieferdecker. “A taxonomy of risk-based testing”. In: *International Journal on Software Tools for Technology Transfer* 16.5 (2014), pp. 559–568. ISSN: 1433-2787. DOI: 10.1007/s10009-014-0332-3.

[29] Roger Ferguson and Bogdan Korel. “The Chaining Approach for Software Test Data Generation”. In: *ACM Trans. Softw. Eng. Methodol.* 5.1 (1996). jan, pp. 63–86. ISSN: 1049-331X. DOI: 10.1145/226155.226158.

[30] Francisco Gutenberg S. Filho et al. “Correlations among Software Testability Metrics”. In: *Proceedings of the XIX Brazilian Symposium on Software Quality*. SBQS ’20. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450389235. DOI: 10.1145/3439961.3439972.

[31] Martin Fowler. *Refactoring. Wie Sie das Design bestehender Software verbessern*. ger. 2nd ed. Fowler, Martin (VerfasserIn). Frechen: mitp, 2020. 1474 pp. ISBN: 9783958459427.

[32] Roy S. Freedman. “Testability of software components”. In: *IEEE Transactions on Software Engineering* 17.6 (1991), pp. 553–564. URL: http://mason.gmu.edu/~kbaral4/Papers/Freedman1991.pdf (visited on 09/02/2023).

[33] Fu Jianping, Liu Bin, and Lu Minyan. "Present and future of software testability analysis". In: *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*. Vol. 15. Oct, 2010, pp. V15-279-V15–284. DOI: 10.1109/ICCASM.2010.5622622.

[34] J. Gao and M.-C. Shih. "A component testability model for verification and measurement". In: *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*. Vol. 2. 2005, 211–218 Vol. 1. DOI: 10.1109/COMPSAC.2005.17.

[35] Vahid Garousi, Michael Felderer, and Feyza Nur Kılıçaslan. "A survey on software testability". In: *Information and Software Technology* 108 (2019), pp. 35–64. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2018.12.003.

[36] Vahid Garousi and Mika V. Mäntylä. "A systematic literature review of literature reviews in software testing". In: *Information and Software Technology* 80 (2016), pp. 195–216. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.09.002.

[37] Boby George and Laurie Williams. "A structured experiment of test-driven development". In: *Information and Software Technology* 46.5 (2004). Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003, pp. 337–342. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2003.09.011.

[38] Mohammad Ghafari, Markus Eggiman, and Oscar Nierstrasz. "Testability First!" In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 1–6. DOI: 10.1109/ESEM.2019.8870170.

[39] GitHub Inc. *GitHub About*. 2023. URL: https://github.com/about (visited on 09/02/2023).

[40] GitHub Inc. *GitHub Octoverse*. 2022. URL: https://octoverse.github.com/ (visited on 09/02/2023).

[41] Liang Gong et al. "DLint: Dynamically Checking Bad Coding Practices in JavaScript". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 94–105. ISBN: 9781450336208. DOI: 10.1145/2771783.2771809.

[42] T. Hariprasad et al. "Software complexity analysis using halstead metrics". In: *2017 International Conference on Trends in Electronics and Informatics (ICEI)*. May, 2017, pp. 1109–1113. DOI: 10.1109/ICOEI.2017.8300883.

[43] Mark Harman. "Refactoring as Testability Transformation". In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. March, 2011, pp. 414–421. DOI: 10.1109/ICSTW.2011.38.

[44] Itti Hooda and Rajender Singh Chhillar. "Software test process, testing types and techniques". In: *International Journal of Computer Applications* 111.13

(2015). URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0fbe1b5515e747025d950658fbc039e98b29b801 (visited on 09/02/2023).

[45] International Organization for Standardization, ed. *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (visited on 03/09/2023).

[46] Muhammad Abid Jamil et al. "Software Testing Techniques: A Literature Review". In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. Nov, 2016, pp. 177–182. DOI: 10.1109/ICT4M.2016.045.

[47] Dalton Jorge, Patricia Machado, and Wilkerson Andrade. "Investigating Test Smells in JavaScript Test Code". In: *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*. SAST '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 36–45. ISBN: 9781450385039. DOI: 10.1145/3482909.3482915.

[48] Irena Jovanović. "Software testing methods and techniques". In: *The IPSI BgD Transactions on Internet Research*, pp. 30–41. URL: http://vipsi.org/ipsi/journals/journals/tir/2009/january/full%20journal.pdf#page=31 (visited on 03/21/2023).

[49] Supaporn Kansomkeat, Jeff Offutt, and Wanchai Rivepiboon. "Increasing Class-Component Testability". In: *IASTED Conf. on Software Engineering*. 2005, pp. 156–161. ISBN: 0-88986-464-0.

[50] Natsuda Kasisopha, Songsakdi Rongviriyapanish, and Panita Meananeatra. "Method Evaluation for Software Testability on Object Oriented Code". In: *2020 59th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*. 2020, pp. 308–313. DOI: 10.23919/SICE48898.2020.9240322.

[51] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. "Software test automation in practice: empirical observations". In: *Advances in Software Engineering* 2010 (2010). URL: https://downloads.hindawi.com/archive/2010/620836.pdf (visited on 09/02/2023).

[52] Satnam Kaur and Paramvir Singh. "How does object-oriented code refactoring influence software quality? Research landscape and challenges". In: *Journal of Systems and Software* 157 (2019), p. 110394. ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.110394.

[53] Pavneet Singh Kochhar et al. "An Empirical Study of Adoption of Software Testing in Open Source Projects". In: *2013 13th International Conference on Quality Software*. July, 2013, pp. 103–112. DOI: 10.1109/QSIC.2013.57.

[54] Aymen Kout, Fadel Toure, and Mourad Badri. “An Empirical Analysis of a Testability Model for Object-Oriented Programs”. In: *SIGSOFT Softw. Eng. Notes* 36.4 (2011). aug, pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1988997.1989020.

[55] Lu Luo. “Software testing techniques”. In: *Institute for software research international Carnegie mellon university Pittsburgh, PA* 15232.1-19 (2001), p. 19. URL: https://ignite.org.pk/wp-content/uploads/2018/12/1388051766_rfp1_Software-testing-techniques.pdf (visited on 09/02/2023).

[56] Matej Madeja et al. “Automating Test Case Identification in Java Open Source Projects on GitHub”. In: *Computing and Informatics Journal* 2021.Vol. 40 No. 3 (2021), pp. 575–605. DOI: 10.48550/arXiv.2102.11678.

[57] Matej Madeja et al. “Empirical Study of Test Case and Test Framework Presence in Public Projects on GitHub”. In: *Applied Sciences* 11.16 (2021). ISSN: 2076-3417. DOI: 10.3390/app11167250. URL: https://www.mdpi.com/2076-3417/11/16/7250 (visited on 02/14/2023).

[58] Tim Menzies et al. “Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle”. In: *Empirical Software Engineering* 22.4 (2017), pp. 1903–1935. ISSN: 1573-7616. DOI: 10.1007/s10664-016-9469-x.

[59] Ali Mesbah and Mukul R. Prasad. “Automated Cross-Browser Compatibility Testing”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 561–570. ISBN: 9781450304450. DOI: 10.1145/1985793.1985870.

[60] Bertrand Meyer. “Seven Principles of Software Testing”. In: *Computer* 41.8 (2008). Aug, pp. 99–101. ISSN: 1558-0814. DOI: 10.1109/MC.2008.306.

[61] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. “Guided Mutation Testing for JavaScript Web Applications”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 429–444. DOI: 10.1109/TSE.2014.2371458. (Visited on 02/14/2023).

[62] Manish Motwani and Yuriy Brun. “Automatically Generating Precise Oracles from Structured Natural Language Specifications”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May, 2019, pp. 188–199. DOI: 10.1109/ICSE.2019.00035.

[63] Matthias M. Müller. “The Effect of Test-Driven Development on Program Code”. In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 94–103. ISBN: 978-3-540-35095-8. DOI: 10.1007/11774129_10.

[64] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The art of software testing. Now covers testing for usability, smartphone apps, and agile development environments*. eng. 3. ed. Hoboken, NJ: Wiley, 2012. 240 pp. ISBN: 9781118031964. DOI: 10.1002/9781119202486.

[65] Morteza Zakeri Nasrabadi and Saeed Parsa. "Learning to Predict Software Testability". In: *2021 26th International Computer Conference, Computer Society of Iran (CSICC)*. 2021, pp. 1–5. DOI: 10.1109/CSICC52343.2021.9420548.

[66] Gerard O'Regan. *Concise Guide to Software Testing*. eng. 1st ed. 2019. Springer eBooks Computer Science. O'Regan, Gerard (VerfasserIn). Cham: Springer, 2019. 84 pp. ISBN: 978-3-030-28493-0. DOI: 10.1007/978-3-030-28494-7.

[67] Jiantao Pan. "Software testing". In: *Dependable Embedded Systems* 5.2006 (1999), p. 1. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=28abbfdcd695f6ffc18c5041f8208dcfc8810aaf (visited on 09/02/2023).

[68] Jihyeok Park. "JavaScript API Misuse Detection by Using Typescript". In: *Proceedings of the Companion Publication of the 13th International Conference on Modularity*. MODULARITY '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 11–12. ISBN: 9781450327732. DOI: 10.1145/2584469.2584472.

[69] Chahna Polepalle, Ravi Shankar Kondoju, and Deepika Badampudi. "Evidence and Perceptions on GUI Test Automation - An Exploratory Study". In: *15th Innovations in Software Engineering Conference*. ISEC 2022. New York, NY, USA: Association for Computing Machinery, 2022. ISBN: 9781450396189. DOI: 10.1145/3511430.3511442.

[70] Felix Redmill. "Theory and practice of risk-based testing". In: *Software Testing, Verification and Reliability* 15.1 (2005). https://doi.org/10.1002/stvr.310 https://doi.org/10.1002/stvr.310, pp. 3–20. ISSN: 0960-0833. DOI: 10.1002/stvr.310.

[71] Pavel Reich and Walid Maalej. "Testability Refactoring in Pull Requests: Patterns and Trends". In: *IEEE/ACM 45th International Conference on Software Engineering (ICSE)* 2023 (2023), pp. 1508–1519. DOI: 10.1109/ICSE48619.2023.00131.

[72] D. S. Rosenblum. "A practical approach to programming with assertions". In: *IEEE Transactions on Software Engineering* 21.1 (1995). Jan, pp. 19–31. DOI: 10.1109/32.341844.

[73] Prateek Saxena et al. "A Symbolic Execution Framework for JavaScript". In: *2010 IEEE Symposium on Security and Privacy*. May, 2010, pp. 513–528. DOI: 10.1109/SP.2010.38.

[74] Marija Selakovic and Michael Pradel. "Performance Issues and Optimizations in JavaScript: An Empirical Study". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 61–72. ISBN: 9781450339001. DOI: 10.1145/2884781.2884829.

[75] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. "Why We Refactor? Confessions of GitHub Contributors". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 858–870. ISBN: 9781450342186. DOI: 10.1145/2950290.2950305.

[76] Danilo Silva et al. "RefDiff 2.0: A Multi-Language Refactoring Detection Tool". In: *IEEE Transactions on Software Engineering* 47.12 (2021), pp. 2786–2802. DOI: 10.1109/TSE.2020.2968072.

[77] Leonardo Sousa et al. "Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 186–197. ISBN: 9781450375177. DOI: 10.1145/3379597.3387477.

[78] Amjed Tahir. "A study on software testability and the quality of testing in object-oriented systems". University of Otago, 2016. URL: http://hdl.handle.net/10523/6143 (visited on 09/02/2023).

[79] Valerio Terragni, Pasquale Salza, and Mauro Pezzè. "Measuring Software Testability Modulo Test Quality". In: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 241–251. ISBN: 9781450379588. DOI: 10.1145/3387904.3389273.

[80] Mark Ethan Trostler. *Testable JavaScript: Ensuring Reliable Code*. O'Reilly Media, Inc, 2013. ISBN: 978-1-449-32339-4.

[81] Mubarak Albarka Umar and Chen Zhanfang. "A study of automated software testing: Automation tools and frameworks". In: *International Journal of Computer Science Engineering (IJCSE)* 6 (2019), pp. 217–225. URL: http://www.ijcse.net/docs/IJCSE19-08-06-011.pdf (visited on 09/02/2023).

[82] J. M. Voas and K. W. Miller. "Putting assertions in their place". In: *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. Nov, 1994, pp. 152–157. DOI: 10.1109/ISSRE.1994.341367.

[83] J. M. Voas and K. W. Miller. "Software testability: the new verification". In: *IEEE Software* 12.3 (1995). May, pp. 17–28. ISSN: 1937-4194. DOI: 10.1109/52.382180.

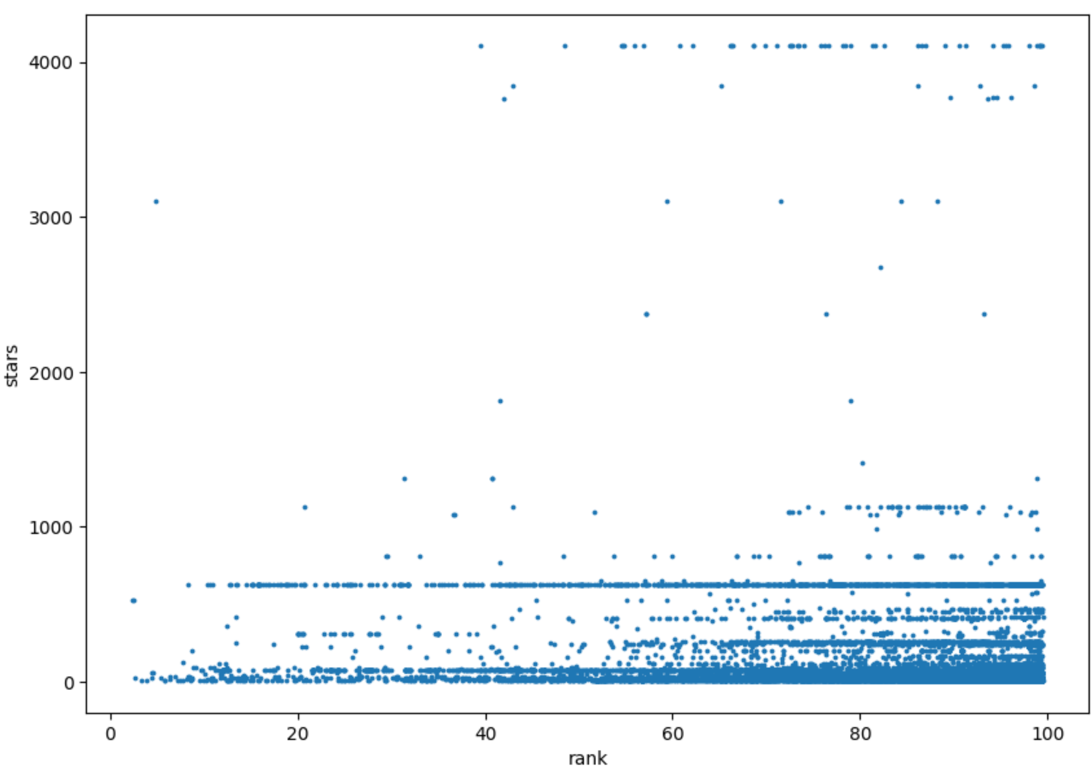
[84] Jeffrey Voas. "Software testability measurement for intelligent assertion placement". In: *Software Quality Journal* 6.4 (1997), pp. 327–336. ISSN: 1573-1367. DOI: 10.1023/A:1018532607070.

[85] Jeffrey M. Voas and Keith W. Miller. "Applying a dynamic testability technique to debugging certain classes of software faults". In: *Software Quality Journal* 2.1 (1993), pp. 61–75. ISSN: 1573-1367. DOI: 10.1007/BF00417427.

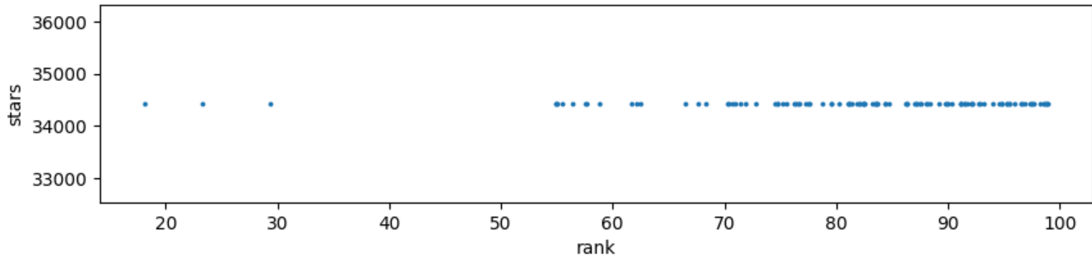
[86] Shiyi Wei, Franceska Xhakaj, and Barbara G. Ryder. “Empirical study of the dynamic behavior of JavaScript objects”. In: *Software: Practice and Experience* 46.7 (2016), pp. 867–889. DOI: `10.1002/spe.2334`.

[87] Chengwei Xiao et al. “Using Spearman’s correlation coefficients for exploratory data analysis on big dataset”. In: *Concurrency and Computation: Practice and Experience* 28.14 (2016), pp. 3866–3878. DOI: `10.1002/cpe.3745`.

Appendix



Appendix 1: Testability level in relation to star rating of repository



Appendix 2: Testability level in relation to star rating for top-rated repo

Table 1: List of testability refactorings

Type	Repo : SHA	File : Line
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 81 AFTER: `js/mc.js` : 106
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 137 AFTER: `js/mc.js` : 153
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 22 AFTER: `js/mc.js` : 10
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 29 AFTER: `js/mc.js` : 4
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 46 AFTER: `js/mc.js` : 77
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 8 AFTER: `js/mc.js` : 21
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 63 AFTER: `js/mc.js` : 94
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 75 AFTER: `js/mc.js` : 181
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 113 AFTER: `js/mc.js` : 133
Internal move	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/mc.js` : 35 AFTER: `js/mc.js` : 67
Add parameter	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/simulation.js` : 3 AFTER: `js/simulation.js` : 3
Add parameter	`Zazcallabah/mce` : `474cb2e1aac9dfcde7af02f8556828ad090e5a20`	BEFORE: `js/simulation.js` : 48 AFTER: `js/simulation.js` : 46

Continued on next page

Table 1 – *Continued from previous page*

Type	Repo : SHA	File : Line
Add parameter	Zazcallabah/mce : 474cb2e1aac9dfcde7af02f8556828ad090e5a20	BEFORE: js/simulation.js : 96 AFTER: js/simulation.js : 94
Add parameter	Zazcallabah/mce : dacc5dfb6e2b96081ade356f87504e03667ae15a	BEFORE: js/storage.js : 27 AFTER: js/storage.js : 53
Add parameter	Zazcallabah/mce : dacc5dfb6e2b96081ade356f87504e03667ae15a	BEFORE: js/page.js : 1 AFTER: js/page.js : 1
Add parameter	jupiter-project/gravity : 1f6cde75ddd2047f1ec0b6b345b48692fc26cebb	BEFORE: config/gravity.js : 879 AFTER: config/gravity.js : 854
Add parameter	jupiter-project/gravity : 1f6cde75ddd2047f1ec0b6b345b48692fc26cebb	BEFORE: config/gravity.js : 879 AFTER: config/gravity.js : 854
Export function	pelias/openstreetmap : 571b6d2009b37bd2e01f3f0f78b29a2f6dde6bef	BEFORE: stream/stats.js AFTER: stream/stats.js : 53
Export function	jupiter-project/gravity : 976d36a7e22683ba1af0898fd50e20b2e88afb3a	BEFORE: src/components/signup.jsx : 8 AFTER: src/components/signup.jsx : 8
Export function	MarioArnt/azure-ad-jwt-lite : b6e152930765df5959b27db36125de79870cf84b	BEFORE: AFTER: src/index.ts : 230
Export function	ably-labs/fully-featured-scalable-chat-app : dd0c52890308a83bcbe700e0038c88e35903a672	BEFORE: api/common/JwtGenerator.ts : 9 AFTER: api/common/JwtGenerator.ts : 16
Export function	ably-labs/fully-featured-scalable-chat-app : d6ffaaebaa33d55c3f1b6d583727585b7ff9f608	BEFORE: api/common/services/UserService.ts : 109 AFTER: api/common/services/UserService.ts : 112

Continued on next page

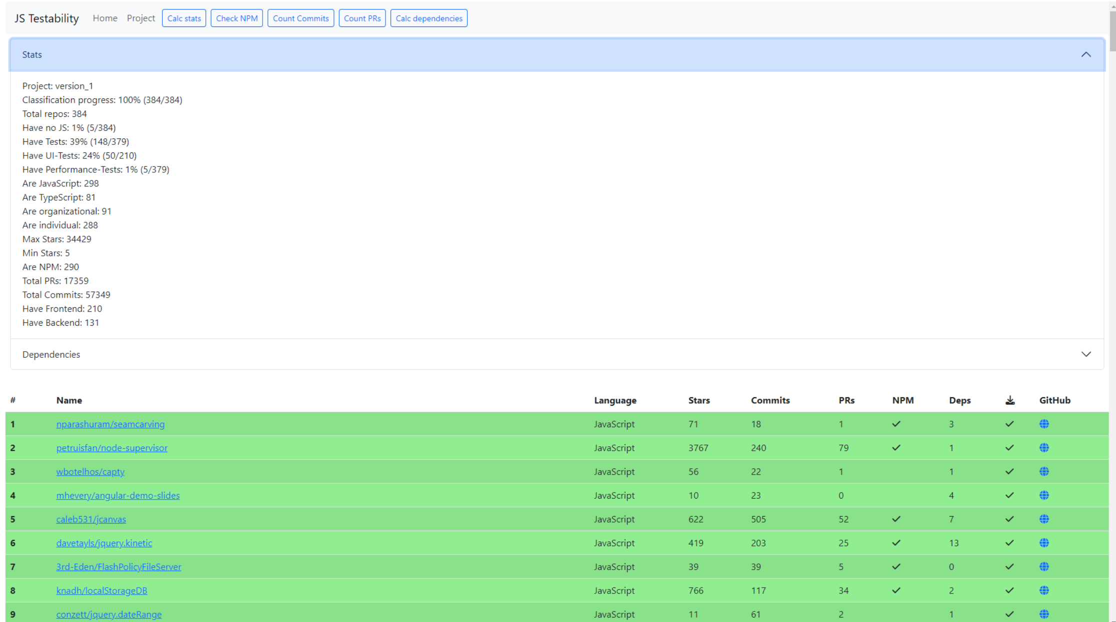
Table 1 – *Continued from previous page*

Type	Repo : SHA	File : Line
Add parameter	auth0/node-saml : b124b6bfa0a40fc8818aba5bc41b5f423bab68f8	BEFORE: lib/saml11.js : 24 AFTER: lib/saml11.js : 25
Add parameter	auth0/node-saml : b124b6bfa0a40fc8818aba5bc41b5f423bab68f8	BEFORE: lib/saml20.js : 24 AFTER: lib/saml20.js : 25
Add parameter	clubajax/HeadlessBrowser : c2e275f037226cc6d745d2e9f1da8bf245d1d5e8	BEFORE: browser.js : 1 AFTER: browser.js : 1
Add parameter	ably-labs/fully-featured-scalable-chat-app : 1dd9eab2b88591789ec2bbf869178b1c5b030fd1	BEFORE: app/src/sdk/BffApiClient.ts : 5 AFTER: app/src/sdk/BffApiClient.js : 3
Set test environment	BreakOutEvent/breakout-frontend : 1b78b5829628d86a900f7270d7e617e309e69252	BEFORE: app.js AFTER: app.js : 38
Set test environment	BreakOutEvent/breakout-frontend : 1b78b5829628d86a900f7270d7e617e309e69252	BEFORE: app.js AFTER: app.js : 214
Set test environment	BreakOutEvent/breakout-frontend : 1b78b5829628d86a900f7270d7e617e309e69252	BEFORE: services/i18n.js AFTER: services/i18n.js : 30
Move rename	Zazcallabah/mce : dacc5dfb6e2b96081ade356f87504e03667ae15a	BEFORE: js/page.js : 153 AFTER: js/viewmodel.js : 49
Move rename	Zazcallabah/mce : dacc5dfb6e2b96081ade356f87504e03667ae15a	BEFORE: js/page.js : 136 AFTER: js/viewmodel.js : 30
Extract	pelias/openstreetmap : a76035d9e2540ad787e446250d1abb1d15fc101a	BEFORE: stream/address_extractor.js : 14 AFTER: stream/address_extractor.js : 5
Extract	SkyZeroZx/API-NestJS-Sky-Calendar : bd6677b4ae5507bac22ae512e5be2e696035412e	BEFORE: src/common/decorators/user.decorator.ts : 3 AFTER: src/common/decorators/user.decorator.ts : 3

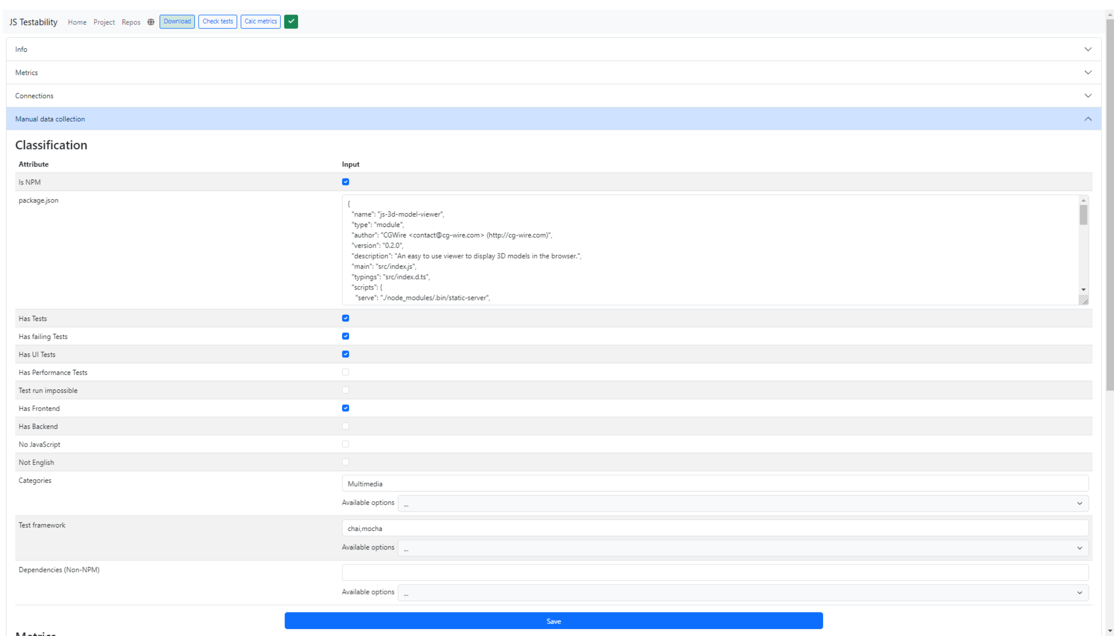
Continued on next page

Table 1 – *Continued from previous page*

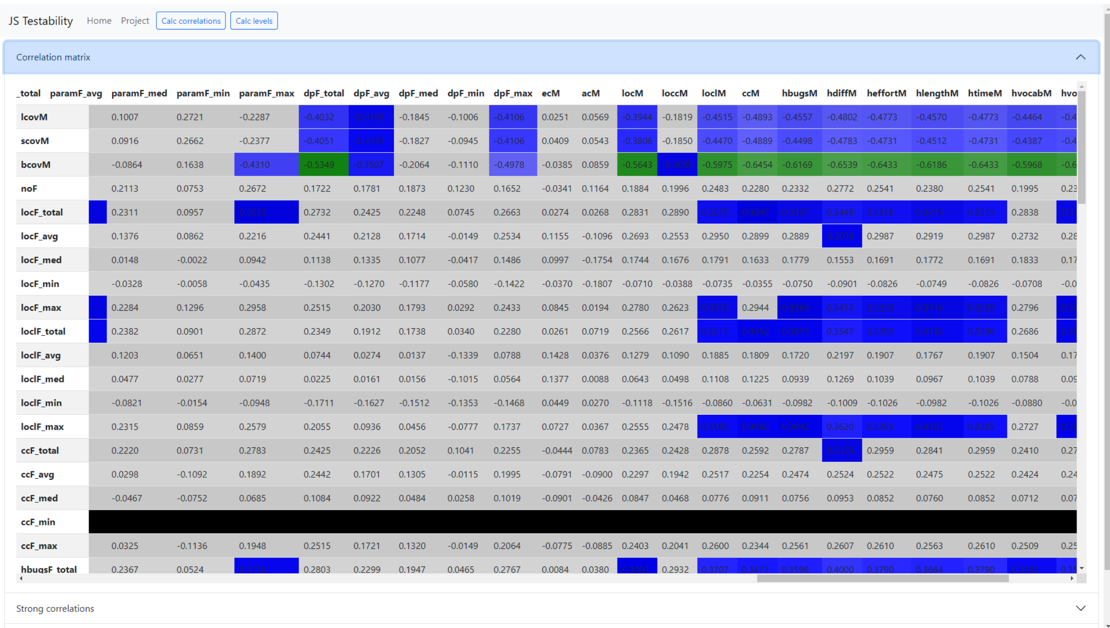
Type	**Repo : SHA**	**File : Line**
Add getter	mapbox/scroll-restorer : 6a46e93db34ae13f80b409ba11b746d7bddbd4e0	BEFORE: AFTER: util/get_window.js : 2
Add getter	Winter22SOFE2720/online-shopping-system : 80b0958deb49e88b308b79fc9e2a5d345843a3cb	BEFORE: Code/src/classes/ShoppingCart.ts AFTER: Code/src/classes/ShoppingCart.ts : 17
Wrap	Zazcallabah/mce : 474cb2e1aac9dfcde7af02f8556828ad090e5a20	BEFORE: js/mc.js AFTER: js/mc.js : 1
Add return value	sheikhmishar/localhost-tunnel : 78fc1e356806c038c348335240cc770ac42a48e4	BEFORE: views/src/js/uiHelpers.js AFTER: views/src/js/uiHelpers.js : 71
Extract	jupiter-project/gravity : 9620c4a94899d7d7cb416b071d2a5462f7ea84fd	BEFORE: config/gravity.js : 1012 AFTER: config/gravity.js : 1011
Add attribute	smswithoutborders/SMSWithoutBorders.com : e1ebdfbbd9eb5e5b7e3b353cbf8637ab54c01b45	BEFORE: src/components/Loader.js AFTER: src/components/Loader.js : 9
Export object	sametpalitci/yelp-clone : 1881cd447fd5b2bdc86844b5428dc74f217fcbf7	BEFORE: AFTER: server/app.js : 28
Move config to object	erukiti/easybooks : 992eff78e886533b74cbce3e3eeaf9af5301da5d	BEFORE: src/build-book/index.ts : 66 AFTER: src/build-book/index.ts : 71
Split module	m-ripper/vuex-multi-history : 150b492e459465fa917dd69b853dbf9489c40da5	BEFORE: src/HistoryPlugin.ts AFTER: src/VuexHistory.ts, src/VuexHistoryPlugin.ts



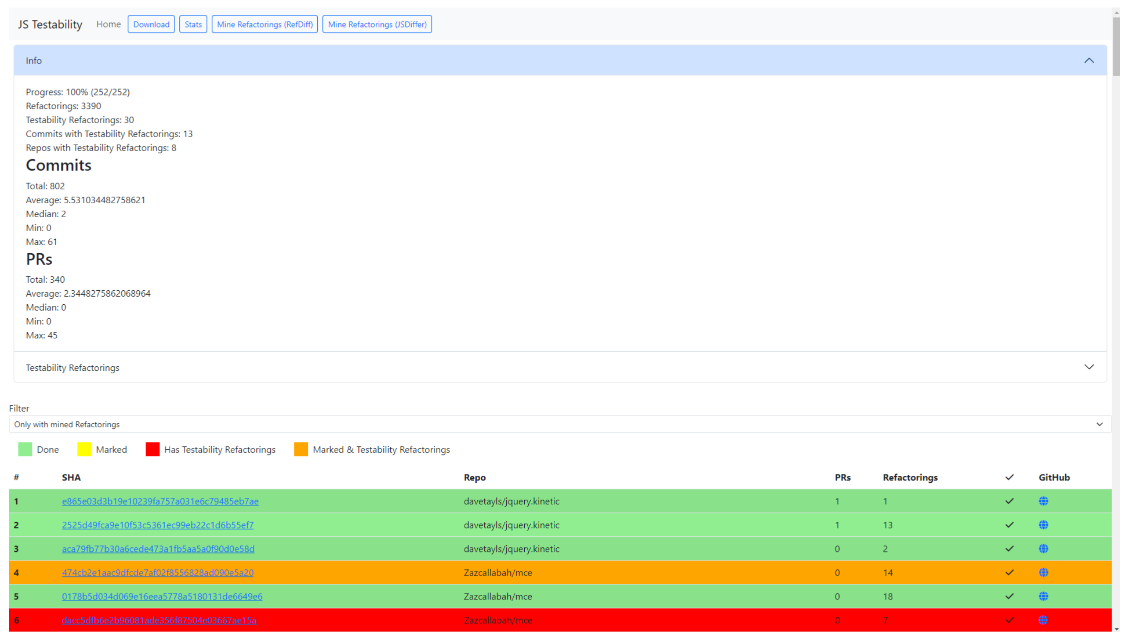
Appendix 3: Screenshot of the repository overview in JSTT



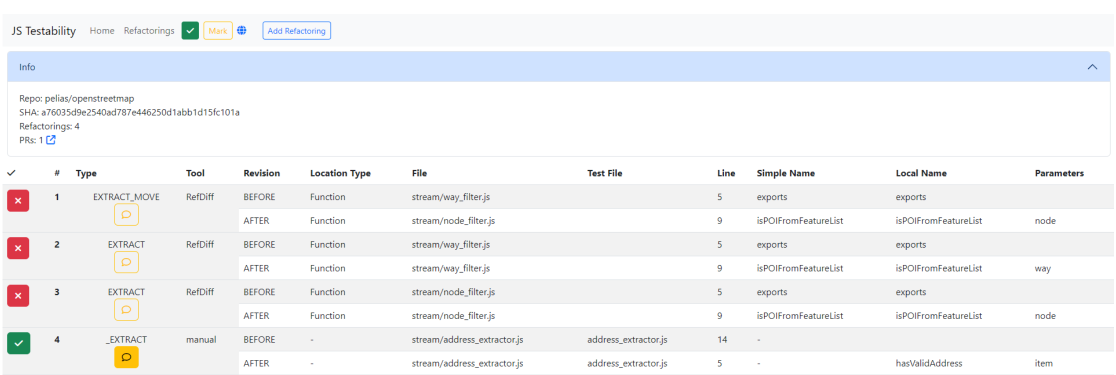
Appendix 4: Screenshot of the repository detail page in JSTT



Appendix 5: Screenshot of the metric correlation matrix in JSTT



Appendix 6: Screenshot of the commit overview for testability refactorings in JSTT



✓	#	Type	Tool	Revision	Location Type	File	Test File	Line	Simple Name	Local Name	Parameters
✗	1	EXTRACT_MOVE	RefDiff	BEFORE	Function	stream/way_filter.js		5	exports	exports	
				AFTER	Function	stream/node_filter.js		9	isPOIFromFeatureList	isPOIFromFeatureList	node
✗	2	EXTRACT	RefDiff	BEFORE	Function	stream/way_filter.js		5	exports	exports	
				AFTER	Function	stream/way_filter.js		9	isPOIFromFeatureList	isPOIFromFeatureList	way
✗	3	EXTRACT	RefDiff	BEFORE	Function	stream/node_filter.js		5	exports	exports	
				AFTER	Function	stream/node_filter.js		9	isPOIFromFeatureList	isPOIFromFeatureList	node
✓	4	_EXTRACT	manual	BEFORE	-	stream/address_extractor.js	address_extractor.js	14	-		
				AFTER	-	stream/address_extractor.js	address_extractor.js	5	-	hasValidAddress	item

Appendix 7: Screenshot of the refactorings list for a commit in JSTT

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

Hamburg, den 03.09.2023

Finn Carstensen

Vorname Nachname

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 03.09.2023

Finn Carstensen

Vorname Nachname