Master's thesis

Master's Programme in Computer Science

Correlation of Unit Test Code Coverage with Software Quality

Juha Tauriainen

May 15, 2023

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Juha Tauriainen

Työn nimi — Arbetets titel — Title

Correlation of Unit Test Code Coverage with Software Quality

Ohjaajat — Handledare — Supervisors

Dr. Antti-Pekka Tuovinen, Dr. Matti Luukkainen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 15, 2023	47 pages, 2 appendix pages

Tiivistelmä — Referat — Abstract

Software testing is an important part of ensuring software quality. Studies have shown that having more tests results in a lower count of defects. Code coverage is a tool used in software testing to find parts of the software that require further testing and to learn which parts have been tested. Code coverage is generated automatically by the test suites during test execution. Many types of code coverage metrics exist, the most common being line coverage, statement coverage, function coverage, and branch coverage metrics. These four common metrics are usually enough, but there are many specific coverage types for specific purposes, such as condition coverage which tells how many boolean conditions have been evaluated as true and false. Each different metric gives hints on how the codebase is tested. A common consensus amongst practitioners is that code coverage does not correlate much with software quality.

The correlation of software quality with code coverage is a historically broadly researched topic, which has importance both in academia and professional practice. This thesis investigates if code coverage correlates with software quality by performing a literature review. Surprising results are derived from the literature review, as most studies included in this thesis point towards code coverage correlating with software quality. This positive correlation comes from 22 studies conducted between 1995-2021, which include Academic and Industrial studies, with studies put into multiple categories, such as Correlation or No correlation based on the key finding, and categories such as Survey studies, Case studies, Open-source studies, based on the study type. Each category has most studies pointing towards a correlation. This finding is in contradiction with the opinions of professional practitioners.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Avainsanat — Nyckelord — Keywords

software testing, code coverage, software quality

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

Software testing is a cornerstone practice of any successful software project, used to verify that the right thing is being built and that the software works as intended. Many types of software testing exist, with unit testing being one of the most popular testing methodologies, in which the smallest details of the software building blocks are tested in isolation [38]. Other methods, such as integration tests or end-to-end tests, verify some of these details as part of their function, however, these other testing methods do not test specific small individual parts of the system, but instead they test how the system at large works together.

The purpose of software testing is not to test the implementation details but instead the output of the software [38]. Software testing should have no opinion about the implementation – when testing is done right, the underlying implementation can be revised or refactored many times without the need to edit the test suites. When done right, software testing can ensure that the implemented features work as intended for as long as possible. This gives safety to the developers and makes it possible to have the personnel and organisational changes around the software project more easily [21, 36, 40].

Testing can give the software developers confidence in their changes, as with relative safety the developers can continue developing the software, knowing that the test cases can catch any unintentional breaking changes [1, 19, 21]. Although it is completely normal for things to break during the development phase, the tests are there to pinpoint these issues. Tests are expected to pass, signalling that the code under test works as intended. A failing test communicates to the developer that a change in the code has introduced some unintentional behaviour to the software, shortening the feedback loop for fixing such an issue [12].

As unit testing tests the smallest parts of the software in isolation, it ensures that these small parts work as intended. Unit tests are often the first tests written for any project and they are cheap and fast to execute, and they should be used in conjunction with other types of tests, such as integration tests, end-to-end tests and system tests. Unit tests are often written alongside the implementation, and for example, in Test Driven Development (TDD) process they are written first and they guide how the implementation is done [12, 21].

Unit tests help developers to more easily test the less often used parts of the software and specific edge cases without much setup needed. If a given functionality is only used in a specific rare case, it might be that some other testing methods, such as the integration tests or the end-to-end tests, might not test these cases. When the unit testing methodology is applied, these specific cases can be tested easily [27].

Developers might have differing views and presumptions on how the code might be used by the system versus how the code will be used by the system. Unit tests might reflect the assumptions the developers have. Real-world use cases and data are often more convoluted than the assumptions and data in the test cases [12, 19, 30].

To gain a better understanding of what has and has not been tested, code coverage can be used for finding the parts which require further testing and to learn which parts are covered by existing test cases [27, 40]. Code coverage is a metric that reports the percentage of the codebase having been executed while running the test suites, as well as indicating which parts of source files are untested through a line-by-line coverage report generated by the test runner. This information can help software developers and testers improve testing on the software. This information is useful, as for example, 70% line coverage on a 100-line file indicates that the test suite has executed 70 lines out of the total 100 lines – with 30 lines being untested, possibly containing unknown issues. With this information, the developers or testers can make decisions on how to improve testing.

Software testing helps improve the software quality, and code coverage is one tool which can be utilised to improve the quality through testing [40]. It is commonly thought amongst the practitioners and contemporary subject matter experts that code coverage has little to no effect on the produced software's quality [6, 12, 19, 21, 22, 40]. This raises an interesting question on what researchers in this field think about the correlation between code coverage and software quality, and how the results differ between academia and industry. Could any conclusions about the quality be drawn on how well the software project is tested?

This thesis presents a literature review containing 22 studies from academia and industry. The results of the thesis are surprising, code coverage seems to correlate with software quality. Code coverage can improve software quality – as having more tests decreases regression and improves confidence in the quality of the software while preventing unintentional bugs from being introduced [1, 19, 21].

The thesis covers academic and industrial studies and both types of studies find correlations between code coverage and software quality in the majority of the studies. The

studies were conducted between 1995-2021, and with 26 years worth of studies, the change in findings is somewhat visible. Interestingly, all the studies which found no correlation between code coverage and software quality were done in or after 2014, although many studies done after 2014 found a correlation.

The thesis is structured as follows. First, in Chapter 2, the key terms, such as code coverage, unit testing, and software quality, are explained. Next, the Chapter 3 explains the methodology used in the literature review that resulted in the studies included in the thesis and how those studies were analysed. The findings from the literature review are opened in detail in Chapter 4. In Chapter 5 the results are discussed. Finally, the thesis ends with Conclusions in Chapter 6.

2 Background

Software testing is an important part of the software development cycle, ensuring for example the maintainability, correctness, and reliability of the software. This chapter opens the key terminology and methods related to software testing and quality, with a focus on unit testing and code coverage, as well as how these two concepts are related to software quality.

2.1 Different Types of Testing

Unit testing verifies in isolation from the rest of the system that an individual unit of code works as specified under different conditions. Unit tests are quick to write and execute, they are small and do not require extensive setup. They are also one the most popular forms of testing [19]. Unit tests focus on the individual units of code – the classes, methods, and functions. The tests are expected to pass.

Unit tests do not usually contain any external dependencies, such as database connections, instead, these are mocked or stubbed. Mocks and stubs are also known as test doubles [16]. Mocking in testing means that the behaviour of a given dependency is simulated, for example, a dependency might perform a long-running action, such as a database call or a network request, which in testing might be unwanted and time or resource-consuming. A stub works in a similar fashion, with the key difference being that it might have some simulated logic to store values in run-time memory. These test doubles are used to decrease the complexity of software testing [16, 38].

If external dependencies, such as database connections, are added to unit tests, they technically stop being unit tests and transform into other types of tests, such as integration tests [8]. In this sense, unit testing is a philosophy and a practice, not strictly a technical concept that can be enforced with automation.

Writing unit tests encourage developers to adopt a more modular architectural approach which will help with maintainability and testability, as well as improve the overall quality of the project. Unit testing provides benefits such as faster bug detection, faster development cycle, and safety to refactor [4, 38].

Automating this process ensures that the given test suite is always executed against the codebase and therefore automated unit tests will have many benefits, such as preventing regression from happening – therefore as a side effect introducing regression testing – in the codebase, as well as preventing any breaking changes to be introduced in the codebase. Automating software testing makes testing more systematic and reliable, while at the same time decreasing costs related to software development, maintenance, testing, and debugging [3]. Implementing the testing frameworks and tests does have its costs, which might end up being significant depending on the size and scope of the project [4].

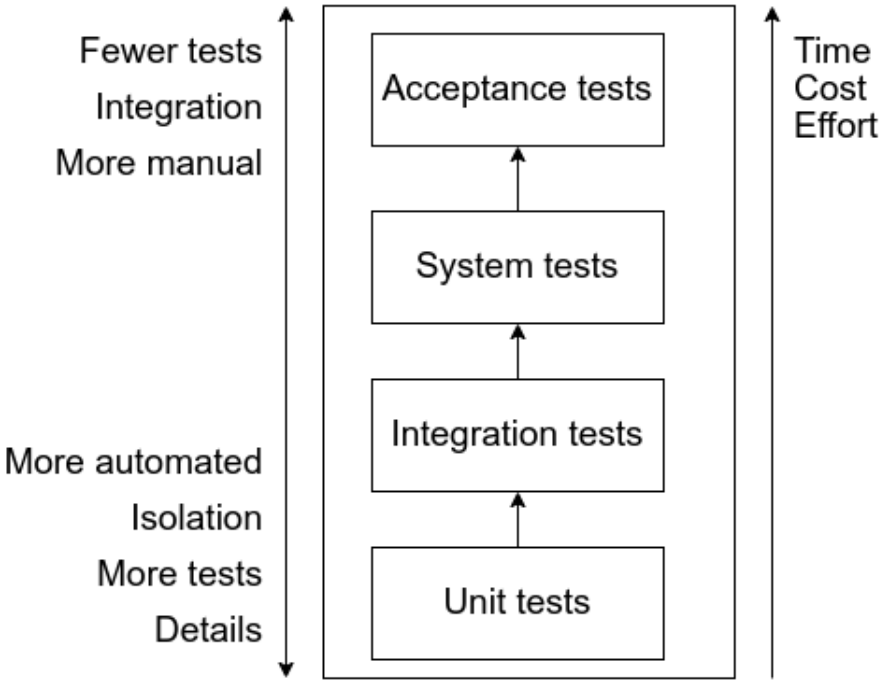

Figure 2.1: Testing hierarchy as described by Vocke in 2018 [42].

Many distinct types of testing for different purposes exist. Some are manual, some are automated, some are more granular, and others are higher level [40]. The hierarchy between different testing methods is illustrated in Figure 2.1. While most testing methods can produce code coverage, unit tests are more commonly used to generate code coverage reports. Depending on what method is used for testing, for example, integration testing can result in many calls happening synchronously and asynchronously throughout multiple threads or processes in parallel, causing the execution to be harder or impossible to track, which results in difficulty to generate a coverage report [12].

Integration testing verifies that the individual parts work together as specified. Integration tests might be passing data between each other as well as to other systems, such as

databases. Integration tests are a bit more complex to set up and run, therefore requiring more time to develop [2].

System testing, or *end-to-end testing*, is a step deeper from integration testing. It verifies that the whole project works together. This type of testing would simulate the end user interacting with the software project in some way. For example, a test case might simulate a user filling in and submitting a form and then expecting a set of results from this interaction [38].

Acceptance testing verifies that the built system is doing the correct thing. Acceptance testing can be automated or manual. Oftentimes acceptance testing is done on new features after the feature is finished but before it is released, to verify that the feature performs the functionality which was required of it. Acceptance testing is a wide concept, usually, the feature is tested with the whole system and verified that it works well with the existing system [38].

Regression testing ensures that no existing features break due to recent changes. All automated tests can therefore be considered to belong to this category. Manual Quality Assurance (QA) testing is also regression testing [38].

2.2 Code Coverage

Code coverage is a measure of how thoroughly the source code has been executed during test cases [35, 47]. Code coverage reports the percentage of code that has been executed in test cases. This percentage can be made out of different coverage metrics, such as line, statement, branch, function coverage, condition coverage, loop coverage, and more. The coverage indicates the parts of the code which have been executed at least once by the test cases and can be used to guide testing efforts [17]. The range of code coverage goes from 0% (not tested at all) to 100% (fully tested). Code coverage reports which parts of the code have been executed (covered) and which have not (uncovered), resulting in a percentage of coverage for different coverage metrics, or a detailed report showcasing the coverage of specific parts of individual files.

Code coverage as a concept has existed since the early 1960s and it has gained more popularity since the late 1990s and early 2000s [14, 24, 35].

Although code coverage may be produced using a variety of different testing methodologies, unit tests are often the testing methodology used to generate coverage reports. Due to

the nature of integration testing or end-to-end testing is typically parallel, with processes being conducted asynchronously, sometimes in parallel threads or in multiple machines or systems, creating code coverage for such testing methodology would be more complicated [38].

The coverage can be calculated by dividing the type of coverage we want to calculate by the total of it in the source code. For example, the line coverage can be calculated by dividing the number of lines that are executed by the overall lines existing in the source code [10, 28].

$$\text{Line coverage} = \frac{\text{Lines executed in a test case}}{\text{Total number of lines}} \tag{2.1}$$

The same formula can be applied to calculate each different type of coverage, for example, branch coverage is branch executed divided by the number of branches in the source code.

Code coverage percentage itself might be somewhat meaningless, which is why it is accompanied by a generated report containing detailed information on what parts of the source code have not yet been tested. Uncovered sections of the source code are highlighted in red, as can be seen in Figure 2.2. The report can also indicate whether a branch has been executed as seen in lines 2 and 3, as in the example test case these if-statements have not been fully covered – values a or b are false in the test case.

```
1     function example2(a: boolean, b: boolean): boolean {
2  1x   E if (a === true) {
3  1x     E if (b === true) {
4  1x         return true;
5            }
6
7            return false;
8          }
9
10         return false;
11    }
12
13    export default example2;
14
```

Figure 2.2: Example of code coverage report with uncovered lines.

Line coverage reports the lines which have been executed during a test case. A single line may contain one or more statements that are executed or they might be partially executed. Line coverage does not care about these statements; if any part of the line was executed during a test case, the line is counted as covered [28].

Statement coverage is a bit different from the line coverage, as a single line might contain multiple statements, such as complex if-statements, value assignments, function, or method calls [38]. In case a single line might have multiple statements, that line might have partial coverage, with parts being highlighted in red if they are uncovered. To fully cover a complex statement, multiple test cases need to be written.

Branch coverage reports how many branches have been executed during a test case at least once [38]. A branch has been covered if the condition on it has been executed at least once by the test suite. Branching occurs when code has control structures such as if-statements and loops that determine which code branch to execute based on certain conditions. Branch coverage is the most difficult of the common coverage metrics to increase due to the complexity involved in branching, making it difficult, but not impossible, to reach full branch coverage.

Condition coverage reports the percentage of all possible outcomes of all conditions which have been executed at least once during the test cases [38]. Branch coverage and condition coverage are similar but differ in key ways. Whereas branch coverage measures how each branch is executed, condition coverage goes a step further and measures how each condition is covered. For example, an if-statement might have three conditions, to gain 100% branch coverage 2 test cases are required: one test for evaluating the if-statement to true and branching into the if-statement block and one to evaluate it to false and continuing execution without branching into the if-statement. For condition coverage test cases must be written for each condition to evaluate either to true or false. One where all values evaluate to true and one where all values evaluate to false. This does not mean that the if-statement itself will ever evaluate as true or false, but full condition coverage for the statement has been achieved. The difference in branch and condition coverage can be seen in Table 2.1.

Step	Line	Statement	Branch	Function	Condition
1	67%	67%	75%	100%	25%
+2	100%	100%	100%	100%	50%

Table 2.1: Different types of coverage metrics for Listing 2.1 with two test cases.

Function coverage indicates what percentage of functions in the code have been executed [13]. A function is considered covered once it is executed in the test cases at least once with any kind of arguments, Function coverage does not tell us how the logic inside the

```
function example1(a, b) {
    if(a == true && b == true) {
        return true;
    }
    return false;
}
```

Listing 2.1: Example 1

```
function example2(a, b) {
    if(a == true) {
        if(b == true) {
            return true;
        }
        return false;
    }
    return false;
}
```

Listing 2.2: Example 2

function might have been covered in the test cases, only that the function itself has been covered.

All these different types of coverage metrics are useful in gaining a deeper understanding of how the code behaves, but the coverage itself does not yet tell us if the code works as intended or is error-free. Even with 100% code coverage, we can not be sure that all the business cases or edge cases have been covered.

Code coverage reports how much of the code has been executed in test cases, which might reflect real-world usage. Full code coverage should not be confused with fully tested code as the coverage can be different depending on the implementation, as can be seen in example functions in Listing 2.1 and Listing 2.2.

Testing the example function in Listing 2.1 with only one test case with provided values of $a = true$ and $b = true$ will result in 67% statement coverage, 75% branch coverage, 67% line coverage 100% function coverage, as seen in Table 2.1. With these values, the if-statement will evaluate to true, and the function execution will branch out into the if-statement scope. This scope has a return statement, which will end the execution of this function, thus the last return statement is never covered. The coverage is visualised in Table 2.3.

When we add another test case which will test the same function with values $a = true$ and $b = false$, we will reach full 100% code coverage, as the if-statement, in this case, will evaluate to false skipping branching out into the if-statement scope, finally executing the return false statement. In this simple example, statement coverage and line coverage go hand in hand, as can be seen in Table 2.1. Even though the line, statement, branch, and function coverage now reaches 100%, condition coverage – and perhaps other less common coverage metrics – is still not 100%.

Given a bit more complex function with a nested if-statement, we would require a bit more

test cases to reach full code coverage. If the example function provided in Listing 2.2 is tested with the same values, $a = true$ and $b = true$, coverage of 60% statement coverage will be achieved, with 50% branch coverage and 60% line coverage. As can be seen, the coverage is now lower than it was when compared to the previous example provided in Listing 2.1.

Adding another test case to the example function in Listing 2.2, such as the second test case in the previous example in Listing 2.1, $a = true$ and $b = false$, will now result in 80% statement coverage, 75% branch coverage and 80% line coverage for the second example function provided in Listing 2.2. The nested if-statement introduced a bit of extra complexity and logic, which prevents the function execution from reaching the end of the function, therefore requiring us to add one more test.

Adding a third test, this time with values $a = false$ and $b = true$, results in the function being tested with most use cases, resulting in 100% coverage for the common metrics. Looking at Table 2.2 we can see that condition coverage is still at 75%. Introducing even simple complexity to the code requires more test cases to be added. Code coverage numbers are not equal, for example, an 80% line statement is easier to achieve than a 75% branch coverage, which can be seen from Table 2.2.

Step	Line	Statement	Branch	Function	Condition
1	60%	60%	50%	100%	25%
+2	80%	80%	75%	100%	50%
+3	100%	100%	100%	100%	75%

Table 2.2: Different types of coverage metrics for Listing 2.2 with three test cases.

Both the function 2.1 and the function 2.2 are now technically fully covered, with 100% coverage on the branches, lines, and statements metrics, but as can be seen in Table 2.3, the function has not been tested with values $a = false$ and $b = false$. This is simple proof that code coverage is not the only indicator developers should pay attention to when writing tests.

When deciding what to test, decisions need to be made on the minimum viable set of tests that cover the implementation logic [2]. The example function in Listing 2.1 could indeed be fully covered with only two unit tests, but would this satisfy a business case, or could this function be considered fully systematically tested?

Code coverage can fluctuate during the development process. A code coverage report can

a	b	Tested in 2.1	Tested in 2.2
true	true	Yes	Yes
true	false	Yes	Yes
false	true	No	Yes
false	false	No	No

Table 2.3: Conditions required to reach full code coverage for Listing 2.1 and 2.2.

be generated alongside test execution, these coverage reports can be used to investigate why code coverage is fluctuating [23]. Fluctuation might happen as new untested code is introduced in the codebase or when a piece of code is removed from the codebase. When code is removed from the codebase, it is common for the code coverage to increase. Similarly, when new code is introduced without additional tests, the code coverage decreases [23, 26].

Measuring the change of coverage allows the developers to get more visibility into the impact of changes they have introduced into the source code [17]. Even though a test case might cover some lines in the code, it does not mean that the said line is free from defects nor tested properly [22]. A project can achieve high code coverage for example by testing only the happy path [18] or by using assertion-free testing [15].

The number of assertions in source code matters, as it has been shown that the overall quality of a given software product is higher when it contains more assertions [46]. Assertions verify that a given function, class, or method returns an expected value. Example of an assertion with the expect.js * testing library is provided in Listing 2.3.

```
expect(example1(true, false)).to.be(false);
```

Listing 2.3: Example of assertion with expect.js library.

The assertions are used in test cases to verify the correctness of a given implementation, with the expected value and output value provided to the assertion. The assertion will then compare these values, if they match, the test case passes, otherwise, the test case fails. A non-passing test case is called a failing test. The failing test signals the developer that either the implementation needs to be changed, or that the test case itself has an issue that needs to be corrected [22, 32, 46].

”Happy path“ is a term used in software testing where the software is tested with a

*https://github.com/Automattic/expect.js/

naive approach, one where the tests are executed in a way that no edge cases or no error handling are executed during testing [7]. The happy path is often also the easiest to implement, as the tester does not need to worry about the edge cases or setting up the tests or infrastructure in a way that would execute each branch of the code. For example, a test case can be given a simple set of steps to achieve the expected output. Real-world scenarios are often quite different. The happy path is often problematic, as it might not take real-world needs into account. Happy path tests are often written by less experienced developers or students, who do not take the complexity and edge cases into account or who might be dismissive toward testing [6, 39].

In the example 2.1 we first tested with a happy path, where both parameters a and b contained a Boolean true value, and then tested with another happy path where a was true and b was false. We did not extend tests beyond the happy path – or from the path of minimal required tests. Tests for example 2.2 as well missed one condition, where both values a and b contained a boolean value false. Even with seemingly full coverage, we might still miss things – such as condition coverage.

Code coverage levels are indicated as percentages, ranging from 0% to 100%. 0% coverage indicates that the codebase does not have any tests and 100% coverage indicates that each part of the codebase has been executed at least once by the test suite [38]. These percentages are not made to be equal though. It takes more time and effort to increase branch coverage than any other coverage metric, whereas line coverage is the simplest to increase. Branch coverage is difficult to increase because branches require more test setup and the code execution branches off into different ways based on how the data flows through the if statements. An if-statement might have multiple conditions which are required to validate to be true or false, which requires extra effort. And not only that, to reach 100% coverage for a single if-statement, the if-statement must be executed at least twice, once where it evaluates to true and code execution continues inside the branch, and once when it evaluates to false and code execution does not continue inside the branch. For the same if-statement, the line coverage is as simple as executing the if-statement just once with any kind of data.

Code coverage percentage below 50% can be considered to be low and would require extra attention from tests. Code coverage between 70% and 85% could be considered to be high and good enough for most purposes [17, 44]. Code coverage of a given software project can reach 100% even when not testing all the features in the codebase. Code coverage only reports whether a given line has been executed in the test suite – not that the said

line was tested intentionally.

Chasing code coverage can be considered a symptom of inexperienced software developers [5, 6, 12]. Junior software developers might consider code coverage as having more importance than it does. In these cases, senior developers should have the responsibility to guide juniors in the right direction [31].

Coverage metrics are abundant. As mentioned earlier, common metrics include *line coverage*, *statement coverage*, *function coverage*, and *branch coverage*. The less common metrics include metrics such as condition coverage, multiple condition coverage, path coverage, entry/exit coverage, loop coverage, and state coverage [38]. Since a multitude of coverage metrics exist, reaching 100% code coverage in tests is near impossible.

2.3 Software Quality

Software quality as a concept can be traced back to 1977, when U.S. Air Force released a handbook on software quality [34]. In this handbook, three key factors were introduced: *Operation*, *Revision*, and *Transition*. Each of these is further opened in the handbook, with the Operation concept focusing on Correctness, Reliability, Efficiency, Integration, and Usability. The Revision concept focuses on Maintainability, Flexibility, and Testability. And finally, the Transition concept focuses on Portability, Reusability, and Interoperability. These main quality concepts still apply. Similarly, in 1993 IEEE released their standard for software quality metrics [25], which include concepts such as Efficiency, Integrity, Survivability, Usability, Correctness, Maintainability, Reliability, Expandability, Flexibility, Interoperability, Portability and Reusability.

Software quality can also be subjective, something being valuable to one person might not be of any interest to someone else. As said by Weinberg in 1992, “Quality is value to some person” [43]. Software projects often have multiple stakeholders, each of whom might consider different things to be valuable. But one of the common themes for quality for each of these stakeholders is testing. Testing creates confidence for the stakeholders, ensuring that their requirements are met and satisfied and that the software works as specified [1, 19, 21].

Freedom from defects, maintainability, correctness, reliability, and reusability are considered common aspects of software quality [23, 25, 34, 43]. Testing can help find these defects and will help keep the defects away, once the defects are fixed and test cases are

written for the fixes. High-quality software is considered to be free from known defects [22, 23, 36]. Quality software works as intended; software must perform well and implement the required specifications. If the software does not solve the issue it was built to solve, it is of bad or low quality. Tests provide insights into the overall state of the project. When a project has no tests, it is usually accompanied by other issues, making it fragile and prone to breaking. In this thesis, quality is handled with a narrow scope, where such aspects as usability, portability, efficiency, and survivability are left out, but instead, each included study can use its own definition of software quality, which is often not mentioned in the source study.

Software quality is a combination of multiple parts – testing is just one part of the formula. Code coverage is a tool that indicates which parts of the code have been tested. Code coverage does not fully indicate quality. Instead, more sophisticated indicators are needed. Testing is hard but necessary. How else could software be verified and validated, if not by testing it thoroughly? And to test a codebase thoroughly, a multitude of tools must be utilised. Unit testing, system testing, integration testing, acceptance testing, and many other types of testing should be used to verify correctness. Software must also fulfill the business requirements and when blindly following code coverage metrics, the tests might not cover the business requirements, making the tests irrelevant or even harmful [38].

3 Methods

In this chapter, the research methods, the research questions, and the steps for analysing the data are introduced. In section 3.3 the research questions are introduced and in section 3.4 the search process is explained.

3.1 Research Questions

This thesis studies the correlation between code coverage and software quality. To align the study, the following Research Questions (RQ) were formulated.

RQ1. What is the current state of research on the correlation between code coverage and software quality in academic and industrial studies?

RQ2. How consistent are the findings on the correlation between code coverage and software quality across different types of studies?

3.2 Search process

Literature Review methodology was utilised to find the source material [41]. Google Scholar was the main search engine to find the studies, with Scopus being used as well. These studies were then accessed from the Institute of Electrical and Electronics Engineers (IEEE) [†] and the Association for Computing Machinery (ACM) [‡] databases.

The topic of code coverage and software quality is well researched, with a basic search query such as "*unit testing coverage*" returning over 1600000 results. Search queries to find the studies included queries such as "*"software testing" "code coverage" quality*" which resulted in 11800 results and "*"software testing" and "code coverage" and "correlation"*" which resulted in 1130 results. Other queries were used as well, such as "*how seniority affects developers +"software testing"*" with 361 results and "*"software testing" "code coverage" quality*" with 249 results at the time or writing.

[†] https://ieeexplore.ieee.org/
[‡] https://dl.acm.org/

From each of these search results, the studies which seemed interesting based on the title, abstract and main findings and which fit the inclusion criteria were chosen for further review. Therefore as studies were picked from different search queries, no single search query was formulated to find the relevant studies.

The initial search queries resulted in a vast amount of results, most of which were not relevant to the study. The initial results guided the search query, which resulted in a set of relevant studies. Studies were filtered based on the inclusion criteria.

Inclusion criteria:

1. The study had to be a peer-reviewed published study.
2. The study is written in English.
3. The study is about code coverage or software quality.

With the inclusion criteria, the literature review search results were filtered to 18 studies. From these 18 studies, further 4 relevant studies were found with the backward snowballing method [45].

These questions were refined by utilising the OpenAI ChatGPT * chatbot, providing it with the thesis title, parts of the introduction chapter, and the key findings. After this, a question was asked from the AI to define possible research questions based on the provided data. The AI provided research questions that were consistent with the sentiments of the research topic and the author's thoughts on the research questions. While unconventional, using AI to refine the questions opens interesting possibilities and is something that needs to be further explored in the academic context. However, an interesting topic on its own, AI-assisted content for this thesis was not utilised further than to refine the research questions.

3.3 How the results were analysed

To find answers to these Research Questions, the search queries above were used and the resulting research papers were filtered based on whether the context of the study was related to code coverage and software quality.

*https://openai.com/blog/chatgpt

Based on the study settings, the studies were grouped into *Academic studies* and *Industrial studies*. The studies were then categorised into three categories based on their findings; *Correlation*, *Slight correlation*, and *No correlation*. Studies were grouped in the "Correlation" category if it was clear that the study indicates a correlation between code coverage and software quality. When the findings were not exact or clear, but the study did find some correlation, the study was placed in the "Slight correlation" category. When it was clear that the study found no correlation between code coverage and software quality, the study was placed in the "No correlation" category.

In this thesis, *Academic study* is a definition for a study that has been conducted in an academic setting, for example, a research paper released by a research group associated with a university, and *Industrial study* is a definition for a study conducted in a non-academic setting, for example as a part of a research group working for a company, such as Microsoft Research. Both the academic and industrial studies included in this thesis conducted their research in companies or outside of companies. Industrial studies might have studied code coverage outside of companies, and academic studies might have conducted the studies in companies.

To further investigate the results based on how the studies conducted their research, the studies were grouped into four additional categories based on the study types and study context, then each category was investigated with the studies included and excluded in those categories. The categories based on the research methods used in the studies are *Case studies* and *Survey studies*. The categories based on which context in which the study is performed are *Studies conducted in companies* and *Open-source studies*. In each category, a comparison is drawn between the studies in the category and those not in that category, to seek an understanding of how the correlation might change between these two groups.

The *Case studies* in this thesis investigate the correlation between code coverage and software quality by investigating testing and coverage tracking in real-world projects or companies, or how university students might utilise testing and coverage reports in software engineering courses [20]. This comparison between real-world projects and university projects brings different insights and findings. The case studies might also utilise other research methods as well, such as the survey method. The *Survey studies* interviewed both open-source developers and professional software developers, either in the context of open-source projects or in a professional context, by utilising either questionnaires or interviews [11]. These interviews could be conducted in companies, amongst open-source

developers, or with university students.

Studies conducted in companies investigated a correlation between code coverage and software quality in the context of real-world projects conducted in large software companies, such as Microsoft, IBM or Google. The *Open-source studies* investigated how the code coverage evolves over time in open-source projects, and they also studied how the code coverage of open-source projects correlates with the number of bugs, issues, and quality in those projects. These projects were investigated by cloning mostly from GitHub. Similarly, these open-source studies could use a variety of research methods. A matrix of the studies with their settings, categories, and findings is available in the Appendix Table A.1.

Two studies were often cited in the studies; *"Test coverage and post-verification defects: A multiple case study"* by A. Mockus et al [S1] and *"Coverage is not strongly correlated with test suite effectiveness"* by L. Inozemtseva et al [S2]. This finding resulted in Figure 4.5, illustrating how the studies are connected.

4 Results

Ref	Year	Categories	Scope	Setting	Correlation
[S1]	2009	CSI-	Microsoft, 6 developers interviewed	Industrial	Yes
[S3]	2002	C---	2 academic studies	Academic	Yes
[S4]	2006	C-I-	Microsoft, 2 case studies	Industrial	Yes
[S5]	2018	C-I-	8 papers reviewed	Academic	Yes
[S6]	2018	C--O	47 open-source projects	Academic	Yes
[S7]	2015	C--O	5 open-source projects	Academic	Yes
[S8]	2015	-SIO	Microsoft, 600+ open-source projects, 210 developers survey	Industrial	Yes
[S9]	2014	C--O	300 open-source projects	Academic	Yes
[S10]	2003	-SI-	Experiment and survey with 24 developers	Academic	Yes
[S11]	2017	C--O	100 large open-source projects	Academic	Yes
[S12]	1995	C---	Studied five programs: Crypt, Uniq, Sort, Grep, Space	Academic	Yes
[S13]	2014	C--O	437 open-source projects	Academic	Yes
[S14]	2019	CSI-	8 developers interviewed, 40 stakeholders, 200 bugs studied	Industrial	Yes
[S15]	2006	C-I-	Studied internal (IBM) 19M line production-level system	Industrial	Yes
[S16]	2016	-S-O	Survey of 212 industrial & open-source developers	Academic	Yes
[S2]	2014	C--O	5 open-source projects	Academic	No
[S17]	2017	-S-O	3 interviews, 2 developers, 1 manager, 2-day workshop	Industrial	No
[S18]	2021	C--O	20 open-source projects in 3 languages	Academic	No
[S19]	2019	C--O	11 open-source projects	Academic	No
[S20]	2021	C---	37 papers reviewed	Academic	No
[S21]	2019	-SI-	Google, 3000 people survey	Industrial	No
[S22]	2017	CSI-	Survey study, 235 survey responses from 7 organizations	Academic	No

Table 4.1: Literature review data.

Table 4.1 contains the studies included in this thesis. This table contains the reference to each study, the year when the study was conducted, the scope of the study, the setting where the study was conducted, the study type categories, and the key findings in regard to correlation. Each study has its corresponding categories listed in the *Categories* column, where *C = Case study*, *S = Survey study*, *I = In company study*, and *O = Open-source study*.

The literature review resulted in 22 studies relevant to this topic. These studies span 26 years, from 1995 to 2021, and they cover the topic well. Overall, 68% of the studies included in this thesis found a correlation between code coverage and software quality. As mentioned in Chapter 3, the findings of the literature review were coded into three

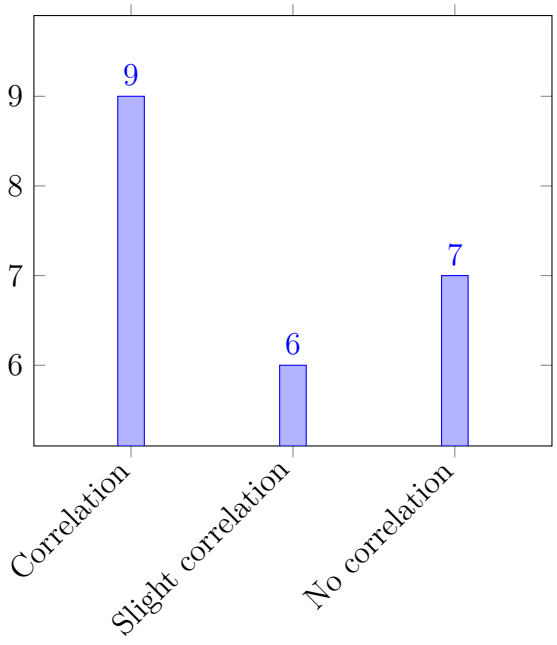

Figure 4.1: Correlation in all studies (22)

categories: *Correlation*, *Slight correlation*, and *No correlation*. Illustrated in Figure 4.1, 9 studies found a correlation between code coverage and software quality, 6 studies found a slight correlation and 7 studies did not find a correlation.

4.1 Correlation found in studies

15 studies found *Correlation* (60%) or *Slight correlation* (40%) between software quality and code coverage, as seen in Table 4.2. 9 studies [S1, S3, S4, S10, S12, S14, S15, S13, S16] found a correlation and 6 studies [S5, S6, S7, S8, S9, S11] found a slight correlation.

As illustrated in Figure 4.2 and Figure 4.3, 9 studies found a Correlation and 6 studies found a Slight correlation. Of the 9 studies, 5 were done in academic settings correlation academic [S3, S10, S12, S13, S16] and 4 studies were done in the industrial setting [S1, S4, S14, S15]. Of the 6 studies finding slight correlation 5 were academic [S5, S6, S7, S9, S11] and 1 was done in industrial setting [S8].

The findings suggest that some of the reasons for the correlation between code coverage and software quality are for example the increase in assertions decreasing defects, testing improving reliability through fewer bugs. The findings also suggest that testing helps in identifying and decreasing bugs, and saves time, especially in long-lived projects which

are released frequently. Testing seems to also decrease code complexity and encourage code decoupling. Also, management plays a key role in making policies such as enforcing testing, which will increase coverage and improve quality.

Ref	Year	Result	Findings
[S1]	2009	Correlation	Correlation between software quality and coverage. They propose code coverage is a sensible and practical measure for test effectiveness.
[S3]	2002	Correlation	Testing increases software reliability and reuse, but required effort increases. Requires a deep understanding of the software.
[S4]	2006	Correlation	Increase in the number of assertions decrease the number of faults.
[S10]	2003	Correlation	TDD practices result in higher code coverage, practitioners believe this increases quality
[S12]	1995	Correlation	Coverage likely to increase software reliability.
[S13]	2014	Correlation	Correlation between widely used coverage criteria and mutation kills.
[S14]	2019	Correlation	Management policy to improve test coverage increased the code quality. Tested code has fewer bugs.
[S15]	2006	Correlation	Code coverage improved the quality of their software project.
[S16]	2016	Correlation	Majority of participants indicated unit testing is important in maintaining the quality of a software project.
[S5]	2018	Slight correlation	Increase in statement coverage might result in a decrease of defects, but this decrease does not seem significant.
[S6]	2018	Slight correlation	Coverage allows developers to have more visibility into the impact of their changes.
[S7]	2015	Slight correlation	Increase in code coverage helps in identifying bugs.
[S8]	2015	Slight correlation	Testing decreases bugs, saves time when software is frequently released over the years.
[S9]	2014	Slight correlation	Increase in code coverage decreases cyclomatic complexity.
[S11]	2017	Slight correlation	Increase in coverage can decrease complexity and coupling. Diminishing improvement in quality.

Table 4.2: Correlations of code coverage and software quality in studies.

4.2 No correlation found in studies

7 studies found no correlation between code coverage and software quality. 5 of these studies were academic [S2, S18, S19, S20, S22], and 2 studies were conducted in industrial settings [S17, S21]. The studies were conducted between 2014 - 2021. Table 4.3 contains

the key findings for the studies which found no correlation between code coverage and software quality and the findings are also illustrated in Figure 4.2 and Figure 4.3.

Findings suggest that coverage might not increase or guarantee software quality, and that test smells can give a false sense of confidence in cases where testing is performed without actually testing the underlying set of features, which can then negatively affect the quality of the tests. Confidence and trust in the test cases and their purpose might then erode away. Three studies mention test smells as being a key reason for no correlation existing between code coverage and software quality. 4 studies explicitly mention that code coverage is no guarantee for software quality.

Ref	Year	Result	Findings
[S2]	2014	No correlation	High coverage does not necessarily increase quality. Coverage helps identify under-tested code, but should not be used as a quality indicator.
[S17]	2017	No correlation	High code coverage does not guarantee good quality. 100% code coverage should not be the goal of software testing, rather than the result of complete testing.
[S18]	2021	No correlation	Rotten green tests give a false sense of confidence, increasing the coverage without actually testing the underlying set of features.
[S19]	2019	No correlation	Study found correlation between test smells and test coverage. Test smells can negatively affect the quality of the tests and influence the test coverage.
[S20]	2021	No correlation	No definitive correlation with tests and software quality – instead correlation with test smells and software quality.
[S21]	2019	No correlation	No definitive correlation – no single reliable measurement for code quality exists. They also mention that code quality is not the only benefit which should arise from code coverage.
[S22]	2017	No correlation	Study found no or weak correlation. No correlation between testing practices and perceived code quality.

Table 4.3: List of studies which found no correlation between code coverage and software quality.

4.3 Academic studies vs Industrial studies

Based on the study settings, the studies are grouped in *Academic* and *Industrial* categories, with 15 academic studies and 7 industrial studies. The data for these are available in Table 4.4, with 67% of academic studies and 71% of the industrial studies finding a correlation between code coverage and software quality. Figure 4.2 illustrates how the results spread

Setting	Studies	Correlation	Count	Percentage
All	22	Yes	15	68%
		No	7	32%
Academic	15	Yes	10	67%
		No	5	33%
Industrial	7	Yes	5	71%
		No	2	29%

Table 4.4: Results by study setting.

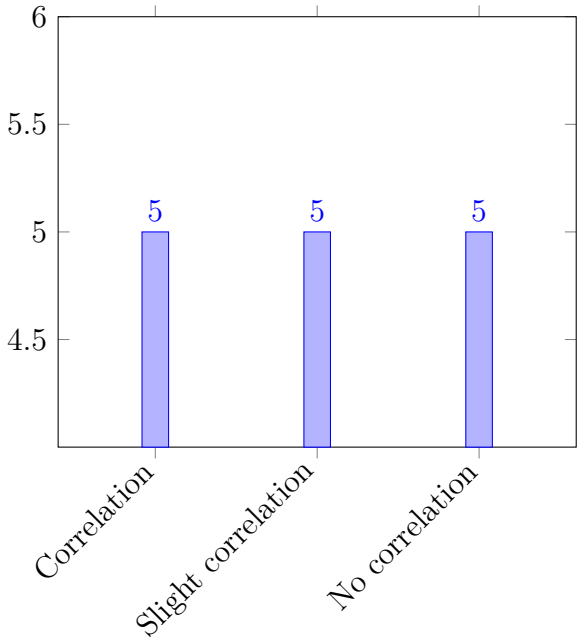

Figure 4.2: Academic studies (15)

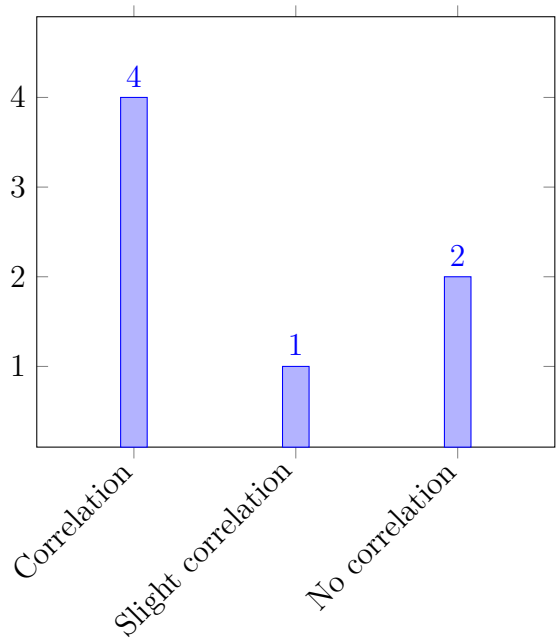

Figure 4.3: Industrial studies (7)

out to the three categories. As can be seen, each category contains 3 studies. Similarly, Figure 4.3 illustrates how the results are spread in the categories. Most of the industrial studies found a correlation between code coverage and software quality.

4.4 Case studies vs non-case studies

Case studies investigated the correlation between code coverage and software quality in the context of real-world projects in companies such as IBM and Microsoft, and open-source projects such as Apache Commons, and in the context of education with student projects and courses. 7 studies were conducted in companies and similarly, 7 studies were

Setting	Studies	Correlation	Count	Percentage
Case studies				
All	17	Yes	12	70,59%
		No	5	29,41%
Academic	13	Yes	8	61,54%
		No	5	38,46%
Industrial	4	Yes	4	100%
		No	0	0%
Non-case studies				
All	5	Yes	3	60%
		No	2	40%
Academic	2	Yes	2	100%
		No	0	0%
Industrial	3	Yes	1	33,33%
		No	2	66,67%

Table 4.5: Case studies vs non-case studies.

conducted in open-source projects. No case studies were conducted at the same time in both company projects and open-source projects. From the case studies which were conducted in companies, 3 found a correlation, and from the open-source case studies, 5 found a correlation between code coverage and software quality.

Case studies are listed in Table 4.5, with 17 studies being case studies and 70,59% finding a correlation between code coverage and software quality. 5 studies were non-case studies and 60% of those found a correlation. 66,67% of the industrial non-case studies found no correlation between code coverage and software quality, being the only category where a majority did not find a correlation.

4.5 Survey studies vs non-survey studies

Table 4.6 contains results for the 8 survey studies, with 3 academic studies, and 5 industrial studies. 5 studies found a correlation, with 2 studies being academic and 3 being industrial. The rest of the studies were non-survey studies, with 71% finding a correlation.

The surveys were conducted in companies such as Google or Microsoft, as well as with

Setting	Studies	Correlation	Count	Percentage
Survey studies				
All	8	Yes	5	62,5%
		No	3	37,5%
Academic	3	Yes	2	67%
		No	1	33%
Industrial	5	Yes	3	60%
		No	2	40%
Non-survey studies				
All	14	Yes	10	71%
		No	4	29%
Academic	12	Yes	8	67%
		No	4	33%
Industrial	2	Yes	2	100%
		No	0	0%

Table 4.6: Survey studies vs non-survey studies.

open-source developers and other stakeholders. The smallest survey consists of three people being interviewed [S17] and the largest was conducted with 600+ open-source developers [S8]. On average, the survey size was about 191 people. The survey data is available in Appendix Table A.2.

Both the survey studies and non-survey studies found a correlation between code coverage and software quality in the majority of the studies. 62,5% of survey studies and 71% of non-survey studies found a correlation. Similarly, when looking at academic and industrial studies in both categories, the majority found a correlation. With academic survey studies 67%, industrial survey studies 60% and in academic non-survey studies 67%, and industrial non-survey studies 100% finding a correlation.

4.6 Studies done in and outside of companies

Table 4.7 lists the 9 studies that were conducted in companies. Most of these studies were performed in the context of a single company, but some [S10, S22] were conducted in multiple companies or universities. 70% of the studies conducted in companies found a correlation, and 75% of the studies conducted outside of companies found a correlation

Setting	Studies	Correlation	Count	Percentage
In companies				
All	10	Yes	7	70%
		No	3	30%
Academic	3	Yes	2	66,67%
		No	1	33,33%
Industrial	7	Yes	5	71,43%
		No	2	28,57%
Outside of companies				
All	12	Yes	9	75%
		No	3	25%
Academic	13	Yes	8	61,54%
		No	5	38,46%
Industrial	0	Yes	0	0%
		No	0	0%

Table 4.7: Studies conducted in companies vs outside companies.

between code coverage and software quality.

5 studies were conducted in large high-profile companies, such as Microsoft [S1, S4, S8], Google [S21] and IBM [S15]. One academic study conducted their research on an unnamed large logistics company in Switzerland with 2500 developers [S14]. Two studies conducted research in multiple companies [S10, S22]. One of these did their research with John Deere, RoleModel Software, and Ericsson [S10] and the other with 6 unnamed companies located in Brazil and Sweden, and with the University of São Paulo in Brazil [S22]. The only two academic studies both studied multiple companies.

7 studies conducted in companies also surveyed their employees or open-source developers [S1, S8, S10, S14, S17, S21, S22]. One study performed in a company setting also investigated open-source projects [S8].

4.7 Open-source studies vs non-open-source studies

Open-source and non-open-source studies are listed in Table 4.8, with 10 studies investigating open-source projects, looking at metrics such as how the code coverage evolves over

Setting	Studies	Correlation	Count	Percentage
Open-source studies				
All	10	Yes	7	70%
		No	3	30%
Academic	9	Yes	6	66,67%
		No	3	33,33%
Industrial	1	Yes	1	100%
		No	0	0%
Non-open-source studies				
All	12	Yes	8	66,67%
		No	4	33,33%
Academic	6	Yes	4	66,67%
		No	2	33,33%
Industrial	6	Yes	4	66,67%
		No	2	33,33%

Table 4.8: Open-source studies vs non-open-source studies.

time in a given project, or how the code coverage of an open-source project correlates with software quality. These studies investigated multiple open-source projects or conducted surveys with open-source developers. From the open-source studies, 70% found a correlation between code coverage and software quality. The majority of these studies were academic, and from the academic studies, 66,67% found a correlation. 100% industrial study found a correlation as well. Only one study was conducted in an industrial setting. When looking at the non-open-source studies, similarly 66,67% found a correlation, in both academic and industrial settings.

The open-source studies which did not find a correlation were all smaller, averaging 12 open-source projects per study, while the studies which found a correlation were larger, averaging 248 open-source projects per study, as well as one open-source survey study which interviewed 212 developers. Five studies had less than 50 open-source projects in their analysis and of these, 2 studies found a correlation, with 5 and 47 projects analysed, while 3 studies did not find a correlation, with 5, 11, and 20 open-source projects being analysed per study.

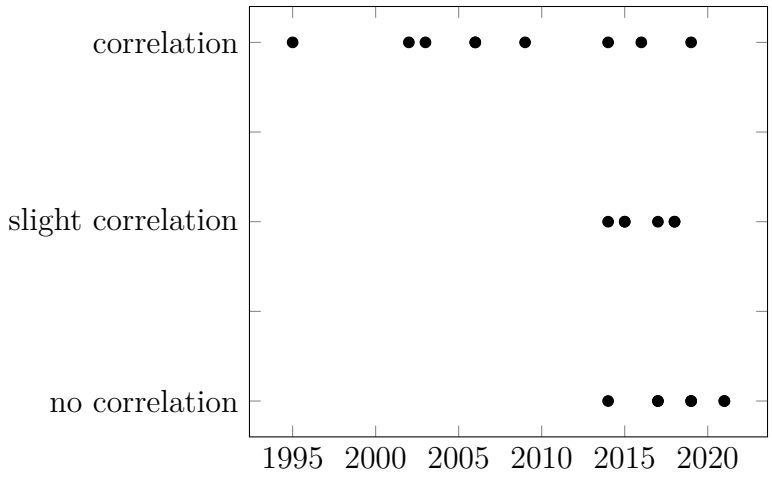

Figure 4.4: Correlation of code coverage and software quality throughout the years.

4.8 Correlation change throughout the years

Figure 4.4 illustrates how the findings have changed over the years. As can be seen, *Correlation* result has been found in studies conducted between 1995 and 2019, *Slight correlation* has been found in studies conducted between 2014 and 2018. Studies resulting in *No correlation* were conducted between 2014 and 2021.

Interestingly the *No correlation* results all were found in or after 2014, while at the same time correlation and slight correlation were found during this same time period. The majority of these *No correlation* studies were done in academic settings. Correlation finding was found steadily throughout the years, with two major year ranges: 2002-2009 and 2014-2019. Interestingly, no results were found for the years 2010-2013.

4.9 Relationships between papers

Figure 4.5 contains the relationships between the literature review studies included in this thesis, displaying how each study refers to the other studies. As can be seen, *"Mockus et al - Test coverage and post-verification defects: A multiple case study"* [S1] and *"L. Inozemtseva et al - Coverage is not strongly correlated with test suite effectiveness"* [S2] are at the core of the studies, with most studies referring to these either directly or through a proxy.

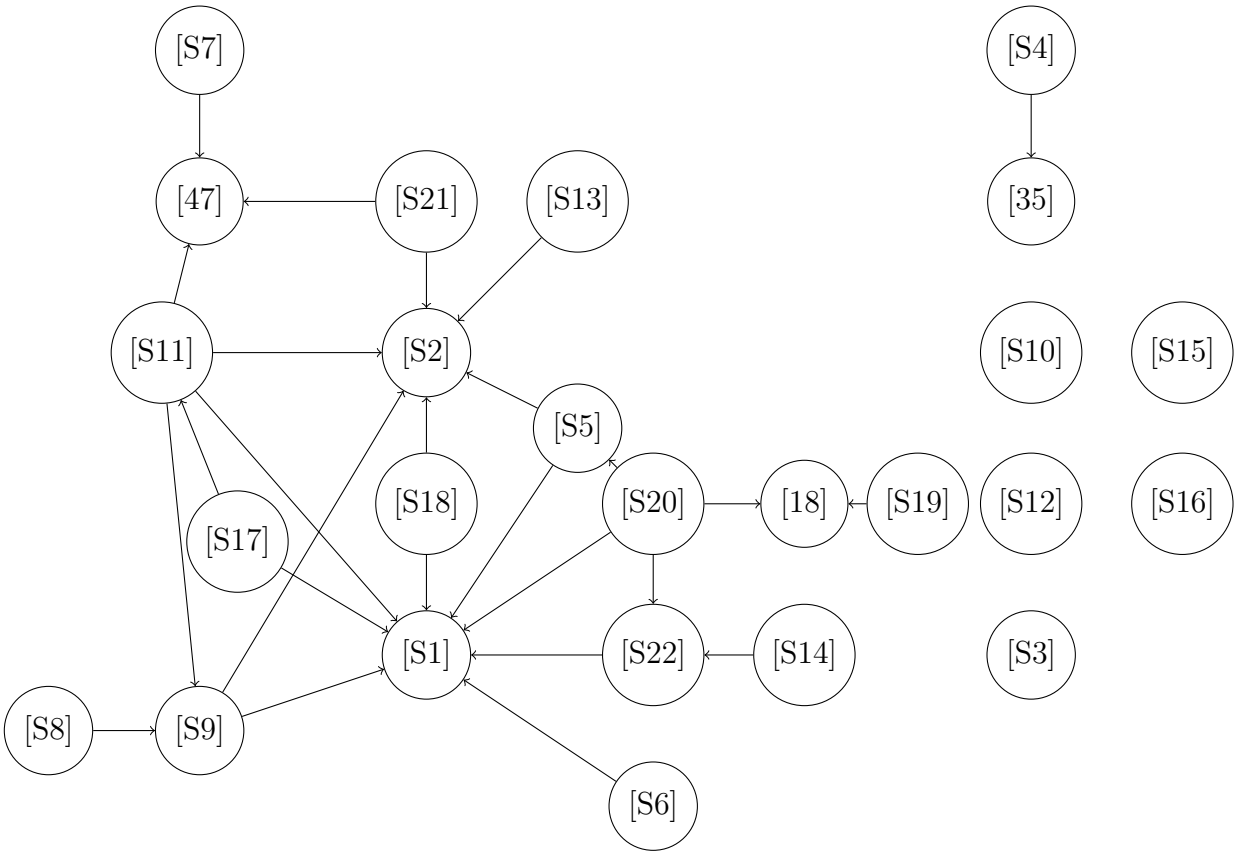

Figure 4.5: Relationships between papers.

5 Discussion

Each metric in this thesis found a correlation between code coverage and software quality. Table 4.4 contains results of the literature review. Results are categorised into two groups, Correlation, and No correlation, and the study setting is categorised into two groups, Academic and Industrial, and their main findings are grouped in *Correlation* or *No correlation*. Of the 22 total studies, 68% found some form of correlation. When looking at Academic studies, the correlation is almost identical, being at 67%. For Industrial studies, the correlation increases a bit, being at 71%. 68% of the studies included in this thesis are Academic studies and 32% are Industrial studies.

5.1 Correlation between coverage and quality

The literature review in this thesis investigated 22 studies done in both academic and industrial settings. The majority of the studies, 15 in total, found some form of correlation between software quality and code coverage. These studies which found correlation were done in both academic and industrial settings, spanning the years 1995 - 2019.

In the case studies, again majority found a correlation. For the non-case studies, the majority of them found a correlation in each setting. For the case studies the data is available in Table 4.5.

When looking at the studies conducted in companies, the majority found a correlation. Similarly, the majority of the studies conducted outside of companies found a correlation, with an interesting data point that no industrial studies were conducted outside of companies. The data for studies conducted in companies is available in Table 4.7.

When comparing studies that investigated open-source projects, the majority of them found a correlation. Same for studies involving non-open-source projects. When looking at the survey studies, again the majority of the studies found a correlation between code coverage and software quality. However the size of the literature review in this thesis must be taken into account, but it does give a good representation of the research done on the topic. Data for the open-source studies is available in Table 4.8.

The open-source studies which found a correlation studied more projects on average than

the open-source studies which found no correlation. On average, the studies which found correlation studied on average 248 open-source projects each, with the smallest sample size being 5 projects and the largest being 600+. The studies which found no correlation studied an average of 12 open-source projects each, with the smallest sample size being 5 projects and the largest being 20 projects. These studies were done between 2014 - 2018. Is such a short time range enough to shift project priorities or change practitioners' minds about testing? Is this just a normal maturing process for long-lived open-source software projects? Also, interestingly only one open-source study was done in an industrial setting.

It is noteworthy that a larger sample size of an open-source project analysed reveals a higher correlation between code coverage and software quality. This finding prompts an inquiry: Could an increased sample size also influence the outcomes of other studies examining the correlation between code coverage and software quality?

And finally, when looking at the survey studies, the majority found a correlation between code coverage and software quality in each setting, with similar findings in the non-survey studies as well. The survey studies data is available in Table 4.6.

5.2 No correlation between coverage and quality

Of the total 22 studies included in this thesis, 7 studies found no correlation between code coverage and software quality [S2, S17, S18, S19, S20, S21, S22]. They have all been published in or after 2014, which raises questions such as what happened in 2014 and why code coverage had more correlation with software quality before this date.

At the same time period, in or after 2014, 9 studies found a correlation [S5, S6, S7, S8, S9, S11, S13, S14, S16], which tells us that we can not easily draw any conclusions on what happened in 2014. These studies were done between 2014 - 2029. Of these 9 studies conducted in this time period, 7 were done in an academic setting and 2 were done in an industrial setting.

Even though only 32% of the studies did not find a correlation between code coverage and software quality, the finding being in a minority of the results does not diminish the fact that they did not find a correlation. These findings were for many reasons, such as coverage not being a guarantee of good quality or freedom from defects [S17], or that increasing coverage might not increase quality [S2], as coverage can be increased simply by adding more tests and not improving code based on the added tests or simply by adding

bad tests, utilising assertion free testing [15] for example, which increases test smells.

Many studies point towards code coverage not being a reliable indicator for the quality of the produced software [S2, S5, S6], or that code coverage has little effect on software quality [6, 19, 21, 40], while at the same time, other studies have coverage to be a useful tool to improve software quality [12, S7]. Code coverage is one tool amongst many which can be used to determine and improve the quality of software. High code coverage does not mean that software is free from defects [S7, 44], and code coverage does not guarantee that all parts of the software have been tested intentionally.

5.3 Correlation does exist

Looking at the results of the included 22 studies, it is clear that code coverage has a correlation with software quality. There is no consensus on how much or how clear the correlation is, but the majority of the data points toward correlation. Many practitioners have opinionated feelings about code coverage when it comes to improving the project [S8]. Mostly these feelings seem to be negative, as coverage is only one metric used for software quality, but coverage should not be underestimated.

If code coverage metrics are tracked and studied, coverage will most likely increase [S15]. The increase might happen due to finding untested code, edge cases, functions, statements, branches, lines, or conditions which has not been covered. A culture of testing is required in each software project, especially long-lived projects [S1, S8]. If an organisation does not have a culture of testing, tracking code coverage metrics will most likely introduce some form of testing culture, which will spread as the impact of testing is understood [S8].

Projects should have some guidelines on coverage metrics, for example, new changes should not decrease overall coverage by too much [S1]. But projects should be careful when enforcing these guidelines as it might lead to unwanted activity, such as gaming the coverage by using assertion-free testing [15]. The purpose of testing software is to ensure that the software behaves as intended and to verify that the correct thing has been built. The point of coverage is to learn which parts of the software require more testing. Looking at coverage will most likely result in new tests or existing ones being improved, which will improve the quality for example by preventing regression.

5.4 Answers to research questions

This thesis set out to answer two research questions.

RQ1: "What is the current state of research on the correlation between code coverage and software quality in academic and industrial studies?"

When comparing academic and industrial studies done on the topic of code coverage correlating with software quality, two things are visible from the data. The first one is that the majority of the studies are academic (15 vs 7) and secondly, the majority of the studies in both settings found a correlation between code coverage and software quality. Figure 4.5 illustrates how the current state of studies is interlinked.

RQ2: "How consistent are the findings on the correlation between code coverage and software quality across different types of studies?"

The findings are consistent. In each setting and each category the majority of the studies found a correlation between code coverage and software quality, 68% of all studies found a correlation, 67% of all academic studies found a correlation, and 71% of all industrial studies found a correlation.

When looking at categories, the consistency becomes more clear; 70,59% of all case studies, 62,5% of survey studies, 70% of all studies conducted in companies, and 70% of all open-source studies found a correlation. The Survey study results seem to indicate a bias that practitioners have against code coverage correlating with software quality, of the four categories it was the one with the lowest result.

The categories were also analysed with studies excluded from each category. For example, the case studies were accompanied by non-case studies. The findings here are also consistent. 60% of non-case studies, 71% of non-survey studies, 75% of studies conducted outside of companies, and 66,67% of non-open-source studies found a correlation between code coverage and software quality.

5.5 The problematic code coverage

Most projects fall short of the 100% code coverage, due to various reasons, such as business-related reasons, time constraints, lack of exposure to tooling, or due to codebase complexity [S8]. Increasing the code coverage is not the goal, instead, coverage increase is just a side

effect of well-tested code [17].

The effort required to reach 50%, 60%, or 70% code coverage is trivial. The only things needed are setting up tests and writing a few initial tests covering common use cases. As test coverage approaches 90% and beyond, the effort and costs related to improving testing and code coverage increase exponentially, which might be hard to justify [S2, S8, S1, 9, 37]. This increase beyond 90% coverage improves fault detection and improves the effectiveness of the test cases.

The earlier defects are detected, the easier, faster, and cheaper fixing them will be. Even though testing will introduce costs in the beginning, in the long term tests will decrease development costs and time. Labour costs include the on-the-job training as well as the actual time spent working on the tests [S10, 19]. High code coverage in actual real projects is uncommon, instead, a code coverage between 70% and 85% is considered good and more realistic [S1, 44].

Changes in code ownership can also have an impact on code coverage. When someone other than the original author modifies the source code, these areas of the code tend to have reduced code coverage [S1]. Outsourcing software development may result in a further reduction in code coverage in specific areas of the codebase.

Testing improves transparency because it signals intent – functionality that has been implemented with thought and care. Testing serves as documentation for the code and the project as well. A new software developer joining the project can inspect and execute the test cases to see how different methods, functions, or classes are used and how they should be used. Organisations might not write tests as they might think that tests will slow down development, or they might see tests as being an unnecessary expense. Organisations might also find it difficult to justify the extra costs of testing, or improving existing test suites to achieve higher code coverage [S1, S2].

Software developers use code coverage metrics as indicators of how well-tested their application is, but as research has shown, code coverage alone is not a good enough indicator of the quality of the application, instead, different kinds of indicators should be used [S5, S6]. One study researched the code coverage evolution of multiple projects, concentrating on the coverage levels of unchanged statements. Their results indicate that a lot of information is lost when only looking at the coverage and that the coverage does not capture the evolution of the codebase [S6].

High levels of code coverage in tests do not indicate whether or not a given test suite is

effective [S2]. In addition to this finding, a complex code coverage may not provide any further information about the test suite to justify the increased costs related to the testing process.

Just looking at code coverage alone is not enough to determine the software quality. Understanding the shortcomings of code coverage and learning how to utilise coverage can help developers in creating better test cases. Using code coverage as a metric for software quality can provide valuable insights into the quality of testing. However, it should be used in combination with other metrics and factors to gain a comprehensive understanding of the overall quality of the software.

5.6 Lack of testing in organisations

Lack of exposure to testing tools has been raised as one of the main reasons why developers do not test their code, and for organisations, time constraints, and compatibility issues have been raised as the main reason for not testing their code [S8, 33, 19]. Software developers who have prior experience in testing, are more open to writing test cases in the future [S4, S8]. The lack of exposure to testing tools could be easily corrected by introducing the software developer to any testing tool, as it is more likely that the software developers with any exposure to tooling, will continue writing tests. These software developers should pick any point from their source code and just start writing tests. As they gain confidence, they can further improve and expand these tests. Therefore to gain exposure, one just has to start using the tools.

The organisation around the software developer should be supportive of this undertaking and should create a culture and atmosphere for testing [S4]. Without a culture of testing in the organisation, it is highly unlikely that testing will occur. As software developers gain more seniority, they gain more insights into the codebase and are capable of writing more useful test cases, therefore lowering the number of defects in the product, which will increase the overall quality of the product.

If the organisation does not have a culture of testing, it is highly unlikely that testing will occur in the organisation, which will lead to more defects and subpar quality [S4]. Therefore the organisation needs to support the testing efforts. Lack of time has been mentioned as one reason for not writing tests; less than 50% of participants responding to a study mentioned that they do not have enough time to perform testing on their code before deploying the code [29]. Some industries, such as banking, finance, and insurance,

are providing software testing training to their staff, whereas other industries are lacking in their training, as their industries, such as IT consulting, might not consider testing to be an important part of their business [9].

5.7 Code coverage effects on quality

Increasing the number of test cases can decrease the number of faults [S4]. Testing is important for applications that are updated regularly [S8] and the coverage metrics give more visibility for developers on the impact of their changes [S6], which can improve the quality [S14, S15]. For projects which do not have a strong testing culture, a change in management policy might be required [S14].

On the other hand code coverage might not necessarily directly increase quality [S2]. Code coverage can also give a false sense of safety, for example, rotten green tests can give a false sense of confidence in the quality, even though tests might not be testing the underlying functionality [S18, S20].

Code quality is not the only benefit that should arise from code coverage [S21]. Full code coverage does not guarantee good quality [S17]. Full code coverage itself is nearly impossible to achieve as all kinds of metrics exist and new ones will appear over time. Instead, each team should decide for themselves what coverage, if any, is reasonable or minimum. Coverage metrics should be tracked but with a grain of salt. They tell us which parts of the code require more attention from tests.

5.8 Validity and limitations

This thesis covers 22 studies conducted between 1995 - 2021, which might not give an accurate representation of the state of research done on the topic of the correlation between code coverage correlation and software quality. Studies which found no correlation between code coverage and software quality were all done after 2014, this result can affect the validity. Utilising Google Scholar as the main search engine to find the studies affects the validity, as it might not include all the relevant available studies.

What must also be taken into account is that different studies might have different definitions – or no definition – for software quality. This possible inconsistency in how the studies define software quality definitely adds a limitation to this study.

6 Conclusions

This thesis investigated the correlation between code coverage and software quality. The investigation was a literature review containing 22 studies spanning 26 years. These studies were both academic and industrial studies, using varying research methods, such as case or survey studies, which resulted in diverse results on the topic. The majority of the studies, 68%, found a correlation between code coverage and software quality.

Two research questions were answered:

RQ1: "What is the current state of research on the correlation between code coverage and software quality in academic and industrial studies?"

The majority of the studies are academic and the majority of the studies in both settings found a correlation between code coverage and software quality.

RQ2: "How consistent are the findings on the correlation between code coverage and software quality across different types of studies?"

The findings are consistent. In each setting and each category, the majority of the studies found a correlation. 68% of all studies, 67% of all academic studies, and 71% of all industrial studies found a correlation between code coverage and software quality. When looking at categories, the consistency becomes more clear; 66,67% of all case studies, 62,5% of survey studies, 66,67% of all studies conducted in companies, and 70% of all open-source studies found a correlation.

The result of this study is contradictory to contemporary domain experts' and professional practitioners' opinions. It seems that it is the opinion of many contemporary subject matter experts – as well as practitioners – that code coverage on its own does not have much value as an indicator of software quality. But the results of this thesis point toward code coverage correlating with software quality. This is a surprising find, especially as nearly 70% of academic and industrial studies found a correlation between code coverage and software quality.

The reality is not as black and white though, code coverage does correlate with software quality in the majority of the studies included in this thesis, while a minority of the studies found no correlation with code coverage and software quality. Even within the studies where the correlation was found, the level of correlation varied. Within the studies

which found a correlation, 60% found a certain correlation, and 40% found only a slight correlation.

While it is true that software testing and code coverage have existed since the 1960s, it took a fair amount of time to gain footing in the professional world. This might explain partly why the change in results. Perhaps sometime in 2014 a threshold on the amount of research was reached, which tipped the results as being negative toward the correlation between code coverage and software quality. Perhaps at this time, new metrics on the correlation or quality were discovered, which raises an interesting question that would require further research to gain a better understanding of why this change has happened, what caused the change to happen, are these kinds of shifts in attitudes and results caused by shifting trends? Something has happened in the last decade which resulted in this paradigm shift. Some of these studies interviewed practitioners, which might have skewed the data due to personal biases from the practitioners. Many studies done between 2014 - 2019 found some form of correlation or improvement in software quality due to increased code coverage, but it must be said that more tests increase coverage and prevent regression. The only way to increase code coverage in tests is to write more tests.

This thesis studied research done between 1995-2021 – with research done in the last decade being the priority – which should give a fair cross-section of the topic during this period. Interestingly, all studies done before 2014 found some correlation between code coverage and software quality. This might be due to biased search terms or filtering of search results in this thesis.

The studies included in this thesis are both academic and industrial, with 5 academic studies [S3, S10, S12, S13, S16] finding correlation and 4 industrial studies [S1, S4, S14, S15] finding correlation. 5 academic studies [S5, S6, S7, S9, S11] found a slight correlation and also 1 industrial study [S8] found a slight correlation. 5 academic studies [S2, S18, S19, S20, S22] found no correlation, as well as 2 industrial studies [S17, S21] finding no correlation between code coverage and software quality.

Quality software will work as intended and quality can mean different things to different stakeholders. Testing is one of the ways we can verify and ensure quality, but far from the only one. Testing will signal quality, and well-designed software is most likely well-tested as well. For the developers, testing uncovers potential issues and might help guide further development of software projects. For managers testing might signal that the software is relatively free from known issues, reliable, and that the developers have given proper consideration to details. For customers, it might mean that the software is stable and

minor details of the software have been given proper care and thought.

Finally, 68% of all studies included in this thesis found some form of correlation between code coverage and software quality. Each study might have defined quality a bit differently and used different research methods, but still interestingly the majority of the studies resulted in the finding that code coverage correlates with software quality. Up to a certain point.

Bibliography

[1] K. Alemerien and K. Magel. "Examining the effectiveness of testing coverage tools: An empirical study". In: *International journal of Software Engineering and its Applications* 8.5 (2014). Publisher: Citeseer, pp. 139–162.

[2] P. Ammann and J. Offutt. *Introduction to software testing*. en. OCLC: ocn180851905. New York: Cambridge University Press, 2008. ISBN: 978-0-521-88038-1.

[3] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron. "Mythical Unit Test Coverage". In: *IEEE Software* 35.3 (May 2018). Number: 3 Conference Name: IEEE Software, pp. 73–79. ISSN: 1937-4194. DOI: 10.1109/MS.2017.3281318.

[4] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. "Test code quality and its relation to issue handling performance". In: *IEEE Transactions on Software Engineering* 40.11 (2014). Publisher: IEEE, pp. 1100–1125.

[5] G. R. Bai. "Improving students' testing practices". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ICSE '20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 218–221. ISBN: 978-1-4503-7122-3. DOI: 10.1145/3377812.3381401. URL: https://doi.org/10.1145/3377812.3381401.

[6] G. R. Bai, J. Smith, and K. T. Stolee. "How Students Unit Test: Perceptions, Practices, and Pitfalls". In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE '21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 248–254. ISBN: 978-1-4503-8214-4. DOI: 10.1145/3430665.3456368. URL: https://doi.org/10.1145/3430665.3456368.

[7] P. Bollen. "BPMN: A Meta Model for the Happy Path". en. In: *Meteor Research Memorandum* (2010).

[8] H. K. Brar and P. J. Kaur. "Differentiating Integration Testing and unit testing". In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. Mar. 2015, pp. 796–798.

[9] F. Chan, T. Tse, W. Tang, and T. Chen. "Software testing education and training in Hong Kong". In: *Fifth International Conference on Quality Software (QSIC'05)*. ISSN: 2332-662X. Sept. 2005, pp. 313–316. DOI: 10.1109/QSIC.2005.57.

[10] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong. "Code coverage of adaptive random testing". In: *IEEE Transactions on Reliability* 62.1 (2013). Publisher: IEEE, pp. 226–237.

[11] R. Czaja and J. Blair. *Designing Surveys*. en. A Sage Publications Company 2455 Teller Road, Thousand Oaks California 91320: Pine Forge Press, 2005. ISBN: 978-0-7619-2745-7 978-1-4129-8387-7. DOI: 10.4135/9781412983877. URL: http://methods.sagepub.com/book/designing-surveys (visited on 04/14/2022).

[12] E. Daka and G. Fraser. "A Survey on Unit Testing Practices and Problems". en. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. Naples, Italy: IEEE, Nov. 2014, pp. 201–211. ISBN: 978-1-4799-6033-0. DOI: 10.1109/ISSRE.2014.11. URL: http://ieeexplore.ieee.org/document/6982627/.

[13] S. Elbaum, A. G. Malishevsky, and G. Rothermel. "Prioritizing test cases for regression testing". In: *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. ISSTA '00. New York, NY, USA: Association for Computing Machinery, Aug. 2000, pp. 102–112. ISBN: 978-1-58113-266-3. DOI: 10.1145/347324.348910. URL: https://dl.acm.org/doi/10.1145/347324.348910 (visited on 04/22/2023).

[14] W. R. Elmendorf. "Controlling the Functional Testing of an Operating System". In: *IEEE Transactions on Systems Science and Cybernetics* 5.4 (Oct. 1969). Conference Name: IEEE Transactions on Systems Science and Cybernetics, pp. 284–290. ISSN: 2168-2887. DOI: 10.1109/TSSC.1969.300221.

[15] M. Fowler. *AssertionFreeTesting*. URL: https://martinfowler.com/bliki/AssertionFreeTesting.html (visited on 07/15/2022).

[16] M. Fowler. *bliki: TestDouble*. URL: https://martinfowler.com/bliki/TestDouble.html (visited on 03/05/2023).

[17] M. Fowler. *TestCoverage*. URL: https://martinfowler.com/bliki/TestCoverage.html (visited on 07/15/2022).

[18] V. Garousi and B. Küçük. "Smells in software test code: A survey of knowledge in industry and academia". In: *Journal of systems and software* 138 (2018). Publisher: Elsevier, pp. 52–81.

[19] V. Garousi and J. Zhi. “A survey of software testing practices in Canada”. In: *Journal of Systems and Software* 86.5 (2013). Publisher: Elsevier, pp. 1354–1376.

[20] J. Gerring. “What is a case study and what is it good for?” In: *American political science review* 98.2 (2004). Publisher: Cambridge University Press, pp. 341–354.

[21] M. Greiler, A. van Deursen, and M.-A. Storey. “Test confessions: A study of testing practices for plug-in systems”. In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 244–254.

[22] H. Hemmati. “How Effective Are Code Coverage Criteria?” In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. Aug. 2015, pp. 151–156. DOI: 10.1109/QRS.2015.30.

[23] M. Hilton, J. Bell, and D. Marinov. “A large-scale study of test coverage evolution”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. New York, NY, USA: Association for Computing Machinery, Sept. 2018, pp. 53–63. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238183. URL: http://doi.org/10.1145/3238147.3238183.

[24] J. C. Huang. “An approach to program testing”. In: *ACM Computing Surveys (CSUR)* 7.3 (1975). Publisher: ACM New York, NY, USA, pp. 113–128.

[25] “IEEE Standard for a Software Quality Metrics Methodology”. In: *IEEE Std 1061-1992* (Mar. 1993). Conference Name: IEEE Std 1061-1992, pp. 1–96. DOI: 10.1109/IEEESTD.1993.115124.

[26] L. Inozemtseva and R. Holmes. “Coverage is not strongly correlated with test suite effectiveness”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 435–445. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568271. URL: https://doi.org/10.1145/2568225.2568271.

[27] M. Ivanković, G. Petrović, R. Just, and G. Fraser. “Code coverage at Google”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 955–963. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3340459. URL: http://doi.org/10.1145/3338906.3340459.

[28] C. Kaner. “Software negligence and testing coverage”. In: *Proceedings of STAR* 96 (1996), p. 313.

[29] M. Kassab, J. F. DeFranco, and P. A. Laplante. "Software Testing: The State of the Practice". In: *IEEE Software* 34.5 (2017). Conference Name: IEEE Software, pp. 46–52. ISSN: 1937-4194. DOI: 10.1109/MS.2017.3571582.

[30] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng. "Test case prioritization approaches in regression testing: A systematic literature review". In: *Information and Software Technology* 93 (2018). Publisher: Elsevier, pp. 74–93.

[31] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. "Understanding the Test Automation Culture of App Developers". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. ISSN: 2159-4848. Apr. 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102609.

[32] G. Kudrjavets, N. Nagappan, and T. Ball. "Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation". In: *2006 17th International Symposium on Software Reliability Engineering*. ISSN: 2332-6549. Nov. 2006, pp. 204–212. DOI: 10.1109/ISSRE.2006.14.

[33] J. Lee, S. Kang, and D. Lee. "Survey on software testing practices". In: *IET software* 6.3 (2012). Publisher: IET, pp. 275–282.

[34] J. A. McCall, P. K. Richards, and G. F. Walters. *Factors in software quality. volume-iii. preliminary handbook on software quality for an acquisiton manager*. Tech. rep. GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

[35] J. C. Miller and C. J. Maloney. "Systematic mistake analysis of digital computer programs". In: *Communications of the ACM* 6.2 (Feb. 1963), pp. 58–63. ISSN: 0001-0782. DOI: 10.1145/366246.366248. URL: http://doi.org/10.1145/366246.366248.

[36] A. Mockus. "Organizational volatility and its effects on software defects". In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 2010, pp. 117–126.

[37] S. Mohanty, A. A. Acharya, and D. P. Mohapatra. "A survey on model based test case prioritization". In: *International Journal of Computer Science and Information Technologies* 2.3 (2011). Publisher: Citeseer, pp. 1042–1047.

[38] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. 3rd ed. Hoboken and N.J: John Wiley & Sons, 2012. ISBN: 978-1-118-13313-2.

[39] R. Pham, J. Mörschbach, and K. Schneider. "Communicating software testing culture through visualizing testing activity". In: *Proceedings of the 7th International Workshop on Social Software Engineering*. SSE 2015. New York, NY, USA: Association for Computing Machinery, Sept. 2015, pp. 1–8. ISBN: 978-1-4503-3818-9. DOI: 10.1145/2804381.2804382. URL: http://doi.org/10.1145/2804381.2804382.

[40] P. Runeson. "A survey of unit testing practices". In: *IEEE software* 23.4 (2006). Publisher: IEEE, pp. 22–29.

[41] H. Snyder. "Literature review as a research methodology: An overview and guidelines". In: *Journal of business research* 104 (2019). Publisher: Elsevier, pp. 333–339.

[42] *The Practical Test Pyramid*. Feb. 2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html (visited on 02/02/2023).

[43] G. Weinberg. *Quality Software Management: Anticipating Change*. nid. 4. Dorset House Pub., 1992. ISBN: 978-0-932633-22-4.

[44] T. Williams, M. Mercer, J. Mucha, and R. Kapur. "Code coverage, what does it mean in terms of quality?" In: *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179)*. ISSN: 0149-144X. Jan. 2001, pp. 420–424. DOI: 10.1109/RAMS.2001.902502.

[45] C. Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering". en. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. London England United Kingdom: ACM, May 2014, pp. 1–10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601268. URL: https://dl.acm.org/doi/10.1145/2601248.2601268.

[46] Y. Zhang and A. Mesbah. "Assertions are strongly correlated with test suite effectiveness". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 214–224.

[47] H. Zhu, P. A. V. Hall, and J. H. R. May. "Software unit test coverage and adequacy". en. In: *ACM Computing Surveys* 29.4 (Dec. 1997), pp. 366–427. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/267580.267590. URL: https://dl.acm.org/doi/10.1145/267580.267590.

Literature Review Bibliography

[S1] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. "Test coverage and post-verification defects: A multiple case study". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. ISSN: 1949-3789. Oct. 2009, pp. 291–301. DOI: 10.1109/ESEM.2009.5315981.

[S2] L. Inozemtseva and R. Holmes. "Coverage is not strongly correlated with test suite effectiveness". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 435–445. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568271. URL: https://doi.org/10.1145/2568225.2568271.

[S3] M. Muller, R. Typke, and O. Hagner. "Two controlled experiments concerning the usefulness of assertions as a means for programming". In: *International Conference on Software Maintenance, 2002. Proceedings.* ISSN: 1063-6773. Oct. 2002, pp. 84–92. DOI: 10.1109/ICSM.2002.1167755.

[S4] G. Kudrjavets, N. Nagappan, and T. Ball. "Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation". In: *2006 17th International Symposium on Software Reliability Engineering*. ISSN: 2332-6549. Nov. 2006, pp. 204–212. DOI: 10.1109/ISSRE.2006.14.

[S5] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron. "Mythical Unit Test Coverage". In: *IEEE Software* 35.3 (May 2018). Number: 3 Conference Name: IEEE Software, pp. 73–79. ISSN: 1937-4194. DOI: 10.1109/MS.2017.3281318.

[S6] M. Hilton, J. Bell, and D. Marinov. "A large-scale study of test coverage evolution". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. New York, NY, USA: Association for Computing Machinery, Sept. 2018, pp. 53–63. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238183. URL: http://doi.org/10.1145/3238147.3238183.

[S7] H. Hemmati. "How Effective Are Code Coverage Criteria?" In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. Aug. 2015, pp. 151–156. DOI: 10.1109/QRS.2015.30.

[S8] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. "Understanding the Test Automation Culture of App Developers". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. ISSN: 2159-4848. Apr. 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102609.

[S9] P. S. Kochhar, F. Thung, D. Lo, and J. Lawall. "An Empirical Study on the Adequacy of Testing in Open Source Projects". In: *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 1. ISSN: 1530-1362. Dec. 2014, pp. 215–222. DOI: 10.1109/APSEC.2014.42.

[S10] B. George and L. Williams. "An Initial Investigation of Test Driven Development in Industry". en. In: (Mar. 2003), p. 5.

[S11] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan. "Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects". In: *IEEE Transactions on Reliability* 66.4 (Dec. 2017). Conference Name: IEEE Transactions on Reliability, pp. 1213–1228. ISSN: 1558-1721. DOI: 10.1109/TR.2017.2727062.

[S12] F. Del Frate, P. Garg, A. Mathur, and A. Pasquini. "On the correlation between code coverage and software reliability". In: *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. ISSN: 1071-9458. Oct. 1995, pp. 124–132. DOI: 10.1109/ISSRE.1995.497650.

[S13] R. Gopinath, C. Jensen, and A. Groce. "Code coverage for suite evaluation by developers". In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 72–82.

[S14] M. Ghafari, M. Eggiman, and O. Nierstrasz. "Testability First!" In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ISSN: 1949-3789. Sept. 2019, pp. 1–6. DOI: 10.1109/ESEM.2019.8870170.

[S15] M. Gittens, K. Romanufa, D. Godwin, and J. Racicot. "All code coverage is not created equal: a case study in prioritized code coverage". In: *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*. CASCON '06. USA: IBM Corp., Oct. 2006, 11–es. DOI: 10.1145/1188966.1188981. URL: https://doi.org/10.1145/1188966.1188981.

[S16] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft. "Automatically Documenting Unit Test Cases". In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2016, pp. 341–352. DOI: 10.1109/ICST.2016.30.

[S17] C. Prause, J. Werner, K. Hornig, S. Bosecker, and M. Kuhrmann. “Is 100% Test Coverage a Reasonable Requirement? Lessons Learned from a Space Software Project”. In: Nov. 2017. ISBN: 978-3-319-69925-7. DOI: 10.1007/978-3-319-69926-4_25.

[S18] V. Aranega, J. Delplanque, M. Martinez, A. P. Black, S. Ducasse, A. Etien, C. Fuhrman, and G. Polito. “Rotten green tests in Java, Pharo and Python”. en. In: *Empirical Software Engineering* 26.6 (Sept. 2021), p. 130. ISSN: 1573-7616. DOI: 10.1007/s10664-021-10016-2. URL: https://doi.org/10.1007/s10664-021-10016-2.

[S19] T. Virgínio, R. Santana, L. A. Martins, L. R. Soares, H. Costa, and I. Machado. “On the influence of Test Smells on Test Coverage”. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. SBES 2019. New York, NY, USA: Association for Computing Machinery, Sept. 2019, pp. 467–471. ISBN: 978-1-4503-7651-8. DOI: 10.1145/3350768.3350775. URL: http://doi.org/10.1145/3350768.3350775.

[S20] F. Pecorelli, F. Palomba, and A. De Lucia. “The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems”. English. In: *Empirical Software Engineering* 26.2 (2021). ISSN: 1382-3256. DOI: 10.1007/s10664-020-09891-y.

[S21] M. Ivanković, G. Petrović, R. Just, and G. Fraser. “Code coverage at Google”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 955–963. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3340459. URL: http://doi.org/10.1145/3338906.3340459.

[S22] L. Gren and V. Antinyan. “On the Relation Between Unit Testing and Code Quality”. In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2017, pp. 52–56. DOI: 10.1109/SEAA.2017.36.

Appendix A Data

Ref	Setting	Case	Survey	In company	Open-source	Correlation
[S3]	Academic	x				x
[S4]	Industrial	x		x		x
[S5]	Academic	x		x		x
[S1]	Industrial	x	x	x		x
[S2]	Academic	x			x	
[S6]	Academic	x			x	x
[S7]	Academic	x			x	x
[S8]	Industrial		x	x	x	x
[S9]	Academic	x			x	x
[S10]	Academic		x	x		x
[S11]	Academic	x			x	x
[S12]	Academic	x				x
[S13]	Academic	x			x	x
[S14]	Industrial	x	x	x		x
[S15]	Industrial	x		x		x
[S17]	Industrial	x	x	x		
[S18]	Academic	x			x	
[S19]	Academic	x			x	
[S20]	Academic	x				
[S21]	Industrial		x	x		
[S22]	Academic	x	x	x		
[S16]	Academic		x		x	x
	Total	17	8	9	11	15
	Percentage	77%	36%	40%	50%	68%

Table A.1: Study settings and categories

Ref	Info
[S3]	1 academic experiment with a practical training course, 22 students.
[S4]	Microsoft, 2 case studies, both part of the Visual Studio codebase.
[S5]	8 papers studied.
[S1]	Microsoft, Windows Vista, 40+ MLOC, team size: 1000+. Avaya 1 MLOC, team size: $\sim$ 100, 6 developers surveyed with 8 questions.
[S2]	Studied 5 open-source projects.
[S6]	Analysed 47 open-source projects.
[S7]	Studied 5 open source projects, 247 faults.
[S8]	Microsoft. Studied 600+ open-source Android projects. 3900 Android developer survey, 83 responses, 600-person internal survey, 127 responses.
[S9]	300 large open-source projects studied.
[S10]	Experimental trials 8 person groups of developers in three companies (John Deere, RoleModel Software and Ericsson). Survey with 24 developers.
[S11]	Studies 100 large open-source projects.
[S12]	Studied five programs.
[S13]	Studied 437 open source projects.
[S14]	A large logistics company in Switzerland. 5 years of historical data. 14102 files and 15500 issues. 40 developers and stakeholders. 200 bugs studied.
[S15]	IBM. Studied internal 19 million lines production-level system.
[S16]	Survey of 212 developers.
[S17]	Tesat Spacecom GmbH. A 2-day workshop where experts were interviewed using a semi-structured interview method. 2 developers spent 2 years trying to achieve 100% code coverage. 3 people were surveyed; 2 developers and 1 manager.
[S18]	Studied 20 open-source projects in 3 languages.
[S19]	11 open source projects studied, collected 21 types of test smells, five different coverage metrics.
[S20]	37 articles studied, Grey Literature Review.
[S21]	Google. A survey of 3000 people, resulted in 512 responses. Analyzed five years of historical data.
[S22]	6 companies from Brazil and Sweden, 1 university from Brazil. A survey study, 235 survey responses.

Table A.2: Study sizes