DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Analysis of the Impact of Static Analysis Tools on the Code Quality of Open-Source Software

Korbinian Quirin Weidinger

DEPARTMENT OF INFORMATICS

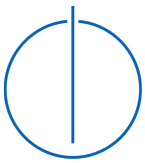
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Analysis of the Impact of Static Analysis Tools on the Code Quality of Open-Source Software

Analyse der Auswirkung von statischen Analysewerkzeugen auf Codequalität von Open-Source-Software

Author:	Korbinian Quirin Weidinger
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Daniel Veihelmann
	Dr. Elmar Jürgens
Submission Date:	August, 15 2021



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, August, 15 2021 Korbinian Quirin Weidinger

Abstract

The necessity of high software quality for the long-term success of software systems is widely known. Resolving quality issues early in the development process leads to less maintenance in the future, less effort in further development, and lower overall costs. Consequently, development teams use audits and code reviews to ensure that developers follow expected coding conventions and find potential bugs early. Integrating a static analysis tool (SAT or linter) to automate such code inspections reduces human effort in code reviews by automatically giving developers feedback on their code without execution. Such feedback ranges from breaking coding conventions to causing potential software bugs or even high-level security threads. Despite their widely accepted benefit, there is little to no empirical evidence for the impact SATs have on code quality.

This paper presents an empirical approach to increase our understanding of the impact of the integration of static analysers on software quality. We define two research questions and a total of eleven hypotheses about SATs and their impact on code quality. We expect projects utilizing SATs to have fewer metric violations than projects in which we can not detect SAT usage. To gain insights on possible impacts, we conduct a large-scale study on over five thousand public GitHub repositories. Using the GitHub REST API, we collect metadata for these projects, such as team size, total commits, or the number of forks. Next, we developed a tool to automatically identify SAT usage within projects by analysing file names and file contents. Since the spectrum of available SATs is vast and language-specific, we focus on detecting a set of eight SATs supporting static analysis for Java, namely Checkstyle, Codacy, CodeQL, Coverity, FindBugs, SpotBugs, PMD, and Sonarqube. The tool identifies if and what specific SAT(s) are integrated into a project with very high accuracy. We use Teamscale to calculate specific code metrics for each repository, including clone coverage, method length, nesting depth, and missing interface documentation. To answer the eleven hypotheses and our research questions, we perform statistical evaluations of our results by analysing the collected data. By proving the validity of our hypotheses, we show that projects that do not integrate static analysers have code metric distributions that are considered worse than the projects using static analysers. We can conclude that static analysers improve overall code quality considering the individual outcomes for all considered code metrics. Lastly, we observe that projects using static analysers are more likely to be popular on GitHub.

The results of this work can give practitioners in software engineering insight on how to assess code quality, a variety of different static analysers for Java, and their impacts on code quality. In addition, our findings may even help development teams pick specific static analysers to improve certain quality criteria based on our results.

Acknowledgments

I want to take this opportunity to express my gratitude to those who made this thesis possible:

First and foremost, I would like to thank my advisor, Daniel Veihelmann. Without his great assistance and dedicated involvement in every step and his previous work, this paper would have never been possible. I want to thank you very much for your support and understanding over these past four months.

I also want to express my gratitude towards my advisor, Dr. Elmar Jürgens, for the warm welcome he gave me at *CQSE GmbH*, for the possibility he gave me to work from abroad as well as for the tremendous amount of research that he was involved in and that has impacted this paper.

Moreover, I would like to thank my supervisor, Prof. Dr. Dr. h.c. Manfred Broy for allowing me to work on this topic.

In addition, I would like to thank my father, who has made it his very mission to teach me the importance of writing high-quality code since the first lines I ever wrote. He taught me to follow coding standards, introduced me to test-driven development (TDD), and made it possible for me to participate in a coding seminar held by Robert C. Martin. Without him, I would not have developed the interests that would later lead to me writing this paper, nor would I have started working for *CQSE*, a company he recommended.

I want to thank my cousin Mathias, a former *CQSE* employee who has had a tremendous impact on my studies at TUM being there as a close friend as well as a mentor during my studies. Lastly, I want to thank the rest of my family for their continuous support before and during my time at university.

Contents

1 Introduction

> *When you're a carpenter making a beautiful chest of drawers, you're not going to use a piece of plywood on the back, even though it faces the wall and nobody will ever see it. You'll know it's there, so you're going to use a beautiful piece of wood on the back. For you to sleep well at night, the aesthetic, the quality, has to be carried all the way through.*
>
> - Steve Jobs, 1985

This chapter explains the motivation behind this work and the challenges one has to face when measuring something as intangible as software quality. In addition, we specify the overall structure of this work.

1.1 Background and Motivation

According to Robert C. Martin, software developers spend over ten times more time reading than writing code [Mar08]. Therefore, developers must adhere to specific coding standards to improve the comprehensibility and maintainability of source code [RT17]. Development teams should seek to identify and resolve quality defects early in the development process when it is cost-effective [Dei+08]. SATs are a way to provide predictions on potential defect causes and can help developers adhere to coding standards during the development process. They reduce human effort in review processes by automating checks for coding standard violations and common defect patterns [Bal13]. It is no surprise that more companies use SATs to analyse code properties ranging from simple coding style rules to more advanced software bugs to multi-tier security vulnerabilities [NWA20]. Due to their simple integration into code collaboration platforms such as GitHub and their execution in automated builds, the use of such tools has also gained popularity in open source software (OSS).

There exists no universal measurement for software quality, and hence it is hard to measure the actual impact SATs have on the quality of a system. One common way to measure software quality is a code metrics analysis [Vei16; Dra96]. Static analysers often use such source code metrics to identify code snippets potentially causing quality issues. The effectiveness of SATs is undisputed, and studies show that teams like to incorporate them into their development process [Aye+08; BC14]. Despite that, we could find little to no evidence that projects incorporating SATs generally have better code quality (metrics) than those that do not.

To understand the real impact of linters on software quality, we conduct an empirical evaluation by performing a large-scale study on OSS systems from GitHub in this work. We will systematically collect relevant data to investigate the quality of software projects as well as the correlation between quality and the use of SATs. In order to measure quality, we define a set of code metrics and thresholds to identify potential software quality issues. The selected metrics will allow us to compare quality aspects between different projects. This way, we can investigate if projects using linters have superior quality compared to other projects.

1.2 Problem Statement

It remains unclear if the use of SATs has a measurable positive impact on code quality. This work aims to examine if SATs have such an impact, and if yes, which quality metrics the use of a SAT affects. Subsequently, reasonably selected software quality metrics are calculated to compare different systems independent of their SAT usage.

Our overall assumption is that SATs have a positive impact on code quality. One more specific assumption would be that projects using SATs have fewer code clones than projects without a SAT. We expect different SATs to impact individual metrics differently, especially since not all SATs generate the same findings; some SATs, for example, support clone detection while others do not.

As mentioned before, the goal is to conduct a large-scale study on OSS systems from GitHub. Our objective requires that we can automatically detect if a SAT is integrated within a system. The correlation between the usage of SATs and the collected software quality metrics is to be investigated using suitable statistic analysis.

1.3 Approach and Goals

As mentioned above, the primary goal of this work is to perform an extensive empirical case study to investigate if we can detect differences in quality metrics among OSS systems integrating versus not integrating SATs. Before conducting our study, we need to determine a maximum set of linters to identify if a linter is used in a project or not. We also need to select a set of code metrics to investigate the code quality of projects. Those metrics will be inspired by the previous work of our supervisor [Vei16] on code quality in OSS. To conduct the previously mentioned large-scale study, we download and analyse a large number of projects from GitHub, a platform hosting millions of OSS systems. The main target is to divide a selected set of OSS systems into two groups,

one group with and the other without SAT integration, to compare their differences in code quality. Our analysis will therefore consist of two steps.

Step one is the detection of SATs within downloaded repositories. Therefore we will develop a tool containing the previously mentioned most extensive possible set of SATs relevant to our study objects to collect data on which SATs are used in which projects. This detection tool will allow us to group our study objects into projects using and not using SATs.

In the second step, we analyse the code quality of downloaded repositories using static analysis to calculate the selected quality metrics. We will obtain the metric measurements from the SAT Teamscale (TS) developed at *CQSE GmbH*.

Using this approach, we obtain an empirical basis that will allow us to evaluate various questions and hypotheses of interest. These questions can be related to SAT usage and the impact the different SATs have on our defined set of code metrics.

This work aims to contribute to research by giving insight into the overall usage of SATs in OSS development and answering more specific questions regarding their impact on software quality. As a result, this thesis may serve as a reference point for the effectiveness of SATs in improving code quality. It might even help developers decide which specific SAT to pick to analyse, tackle and improve certain quality-related aspects in their project.

1.4 Structure of This Work

Chapter *1 Introduction* introduces the topic of this thesis and motivates why we analyse the impact of SATs on software quality in OSS systems. Chapter *2 Fundamentals* proposes our key software quality metrics (*2.1 Software Quality*), provides an overview of SATs relevant to this work (*2.3 Static Analysis Tools*), gives additional background knowledge about other topics relevant for our study, and lists some related works already available. Chapter *3 Case Study Design* explains the approach and execution of our study. Section *3.2 Research Questions* introduces three research questions that we aim to answer by investigating nine hypotheses. In section *3.5 SAT Detection Tool* we describe how we implemented the automatic detection of SAT integration for our study objects. Section *3.6 Code Quality Analysis Procedure* explains the evaluation of proposed code metrics using TS. Chapter *4 Case Study Results and Interpretation of Findings* contains the empirical evaluation of the collected data and the interpretation of our results. In Chapter *5 Threats to Validity*, we will discuss the internal, external, and structural threats to our approach. Chapter *6 Conclusion and Outlook* summarizes our

work and discusses opportunities for future research related to the topic.

2 Fundamentals

This chapter contains background knowledge that is relevant for the rest of this work. We introduce measures of software quality and explain the open-source resources utilized in the course of this study. Also, we provide some background knowledge about SATs and list and describe all linters relevant to this study briefly.

2.1 Software Quality

This section provides an overview of the metrics used to compare the software quality of the different systems inspected in our study. It motivates why we choose these metrics and proposes the metric thresholds used in our analysis.

Measuring Software Quality

Measuring Software Quality is a complex subject as its perception highly depends on the expectations towards a product by the user or developer [RT17]. Also, it remains quite challenging to measure a system's quality as there is no general universally applicable definition of the term software quality [Nak+16].

A study on quantitative evaluation of software quality outlines that the utility of the software from the user's side comes with reliability and efficiency; in contrast, from a development standpoint, it comes with maintainability [BBL76]. The study further divides maintainability into testability and understandability [BBL76]. It is evident that software quality is a crucial factor for the success of a system to keep users and developers satisfied [Nak+16]. The code quality, speaking the maintainability of the code, is therefore only of visible importance for the development process. But low code quality can lead to a less reliable and efficient product, resulting in lower customer satisfaction and overall software quality. Just like software quality, code quality is purely subjective. To compare the quality of different projects, we investigate code quality as it is hard to measure aspects like reliability and efficiency without user surveys.

To measure code quality in a system, we investigate its source code. Since the 60s and 70s, metrics like lines of code (LOC) and Cyclomatic Complexity (CC) became ways to measure code quality aspects [HWY09; FN00]. Development teams used, and still use, LOC to calculate expected efforts, programmer productivity and describe the defect density (number of defects per thousand LOC (KLOC)) [FN00].

We can define threshold values for each code quality metric; these depend on current trends, preferences, programming language, and the application objective. We

assess the metrics clone coverage, nesting depth, file size, method length, and interface documentation in this work.

Software Quality Metrics

To analyse and compare the quality of different software systems, we need to pick a set of code metrics we consider as good quality indicators. We differentiate between three types of metrics. A *size metric* indicates the size of a system by counting the number of files, LOC, source lines of code (SLOC), statements, or methods and is not used to measure the quality of a system but to compare systems in size and to calculate other metrics. A *numeric metric* is a metric that consists of a single numeric value. An *assessment metric* divides the system into parts differentiating between green, yellow, and red assessment (g, y, r) using different size metrics. We define two thresholds, one for yellow and one for red for each assessment metric. To enhance statistical comparability, we convert assessment metrics into *percentage assessment*s. The percentage assessment is defined as follows:

$$(g_p, y_p, r_p) = (\frac{g}{g+y+r}, \frac{y+r}{g+y+r}, \frac{r}{g+y+r})$$

There exists a variety of metrics to compare the complexity and quality of software systems. Göde *et al.* propose that comparing systems based on a single number is problematic [GHJ12]. Hence we do not aim to assess quality with a single value describing the complexity of the entire system; therefore, we consider multiple metrics individually. Most of these metrics were already considered by Veihelmann as well as in other papers [Vei16; JDH10]. We consider the following metrics:

1. Clone coverage
2. Comment completeness
3. Nesting Depth
4. Method Length
5. File Size

We will introduce each metric in detail in the subsequent paragraphs.

Code Duplication

We can find duplicated code (also called *code clones*) in significant amounts in many programs [Jue+09]. Code clones are harmful for two reasons: (1) they increase maintenance cost, and (2) inconsistent changes to cloned code can create faults that can cause incorrect program behavior [Jue+09].

Clone coverage is a value metric and one of the standard cloning metrics that has been around since the early days of clone research [GHJ12]. We use the following definition of clone coverage according to Jürgens in [Jue11]:

$$\text{clone coverage} = \frac{\text{cloned statements}}{\text{overall statements}}$$

The threshold *minimal clone length* highly influences the *clone coverage* of a project. The *clone coverage* of a project is highly influenced by the threshold *minimal clone length*. The *minimal clone length* describes the number of duplicated statements required to consider a piece of code a code clone. Veihelmann and Jürgens chose a minimal clone length of 10 in their studies, which we adopt [Vei16; Jue+09]. Smaller values are often inaccurate to point at quality problems, and larger figures reduce the (observed) clone coverage [Jue11; GHJ12].

We differentiate between four different types of clones, adopting the classification considered by Jürgens [Jue11]:

- type 1 - Copied and pasted code where formatting, white space, or comments may be changed.
- type 2 - Additionally to the type 1 properties, parameters or variables may have different types or names.
- type 3 - Additionally to the type 1 and 2 properties, statements may be inserted or deleted.
- type 4 - Pieces of code with nearly identical functionality even though their implementation is different.

Research shows that the detection of type 3 clones is complex and that the runtime of current algorithms is exponential [Wah+04]. We focus on detecting clones of type 1 and 2 in the course of this work.

Documentation

The documentation of source code, especially of programming interfaces[1], is essential for using and maintaining software components [SDZ07]. Comments can be helpful to clarify and explain concepts to developers, allowing them to understand code faster [Mar08]. They are the primary source of documentation after the code itself [SAO05]. Bad, misleading, or missing documentation can lead to projects being challenging to maintain and extend since original developers may no longer be available [SDZ07]. Not all comments impact understandability; some may be outdated or just commented out code [HKV07]. Thus simply measuring the amount of commented lines of source code is generally not a good indicator for high software quality.

Since currently the only means of assessing the quality of in-line documentation is performing time-consuming manual code checks [KWR10], we focus our quality analysis on interface comments only. As most of the interaction with interfaces happens through method calls, method Javadoc is especially important [SAO05]. We determine if crucial interface comments are present or not, excluding comments for trivial cases such as simple *getter* and *setter* methods while expecting them for all public types, methods, and attributes.

To measure the documentation of a system, we use the value metric *comment completeness*, which Veihelmann defined as follows [Vei16]:

$$\text{comment completeness} = \frac{\text{actual interface comments}}{\text{expected interface comments}}$$

We then compare systems using *missing interface comments = 1 - comment completeness*. Values range from 0% to 100%, where 0% is a perfectly documented system while 100% implies an undocumented system.

It can be pretty challenging to assess the actual quality of a comment itself. Java interface documentation should be written as Javadoc comments since they are a widely known and accepted documentation standard [SK21]. With Javadoc, a JDK tool, documentation can be created automatically from Java source code mostly relevant to external developers that need to use the software API [Abt20; SK21]. In research Javadoc comment quality for methods is sometimes measured by checking the comment for specific Javadoc annotations like *@param*, *@return*, and *@throws* [SDZ07]. Analysing Javadoc annotations allows generating individual scores for Javadoc by dividing the documented items through the documentable items [SDZ07]. In our opinion measuring quality by expecting Javadoc comments to contain documentation for each documentable item can

[1] we use the term *interface* in its broadest meaning, that is, including classes, methods, enums, etc.

result in trivial documentation. Comment quality is difficult to evaluate in an automated way. Hence we only consider *comment completeness* in this work.

Nesting Depth

As developers spent most of their time reading code, the complexity of methods plays a significant factor in quickly understanding its functioning and implementing changes [Mar08; Vei16]. A low level of nesting can be considered an indicator for less complex source code [Fak+19]. For this reason, it is desirable to keep the nesting depth below a particular maximum value [Vei16]. To reduce the nesting of a source code fragment, it is often necessary to refactor code by moving part of the contents of the affected method into a new function [Vei16].

The nesting depth of a method is the number of nested statement blocks due to control structures like branches and loops. In Java, the nesting depth of a method is equal to the maximum depth of the symmetric brackets. We evaluate nesting depth as a method-based assessment metric. We consider nesting starting from our yellow threshold of 4 as deep nesting and above and including our red threshold of 5 as very deep nesting. These values are consistent with similar work that evaluate nesting depth as a quality indicator [Vei16; Sch+15].

To illustrate how this method-based assessment metric works, we consider a small system with 10 methods. We have 6 methods with a nesting depth of 2, 3 methods with a nesting depth of 4, and 1 method with a nesting depth of 6. Therefore, according to our thresholds, we consider 6 methods to have good nesting, 3 methods to have deep nesting, and one method to have very deep nesting, resulting in the nesting assessment $(g, y, r) = (6, 3, 1)$. This leads to the percentage assessment $(g_p, y_p, r_p) = (60\%, 40\%, 10\%)$. These relative values can then be used to compare multiple systems of different sizes.

Method Length

Methods are the smallest executable units that can be executed and tested individually [HKV07]. There are many reasons why it is considered a best practice to keep methods as short as possible, including better understandability, the embodiment of a single meaning, or the ability to fit on a computer screen easily without scrolling [HKV07; Mar08]. Extremely long methods are challenging to understand and modify and might also interfere with the ability of developers to test and reuse their functionality properly [HKV07]. Methods that exceed an accepted length threshold can thus generally be considered as a signal for a potential quality issue. There are some rare exceptions where a long method is considerably quality-friendly, such as a single, lengthy, easy-to-

understand switch statement. We can shorten long methods by extracting subroutines into new methods (*method extracting*) [WKK07]. Many IDEs provide a comfortable way to extract methods, but it can still be challenging for developers to pick the parts of the method that should be extracted [Vei16].

We evaluate method length as a statement-based assessment metric, differentiating between acceptable (green), long (yellow), and very long (red) methods. In literature, thresholds vary from 20 to 200 (source) LOC [Mar08; McC76]. Our chosen thresholds are 30 statements for yellow and 50 for red; accordingly, we add the statements of methods with 30 to 49 statements to y and with 50 and above to r. These thresholds are the boundaries mentioned earlier, correspond to practical considerations, and are consistent with related work evaluating method length as an indicator for code quality [Vei16; Sch+15; Mar08].

To illustrate how this statement-based assessment metric works, we consider a small system with 2 methods. One method has 20 statements while the other has 80. Accordingly we have our method length assessment of $(g, y, r) = (20, 0, 80)$, resulting in the percentage assessment $(g_p, y_p, r_p) = (20\%, 80\%, 80\%)$.

File Size

When evaluating method length as a quality indicator, the primary considerations also apply when considering the file sizes within programs [Vei16]. Long files (measured in SLOC) have more statements and, therefore, often more specific features making them harder to understand, thus leading to higher effort to change [HKV07; Mar08]. A file should contain a clearly defined set of features; very long files can indicate too many features [Mar08]. For this reason, we investigate the file sizes of the projects analysed in this paper and use them as an indicator of quality. While Martin considers 200 LOC as a target goal for systems, thresholds like 300 to 400 SLOC seem more realistic [Mar08; Sch+15; Vei16]. In Java, the file size in SLOC correlates to the class size when we consider having only one top-level class per file.

We evaluate file size as a SLOC-based assessment metric. We choose our yellow threshold to be 300 SLOC and our red to be 500, aligning with other works that consider file size a quality indicator [Vei16; Sch+15].

To explain how this SLOC-based assessment metric works, we take a system with three files of lengths 100, 300, and 600 SLOC into consideration. According to our thresholds we acquire an assessment of $(g, y, r) = (100, 300, 600)$, resulting in the percentage assessment $(g_p, y_p, r_p) = (10\%, 90\%, 60\%)$.

2.2 Open-source Software

OSS is released under a license in which the copyright holder grants everyone free access to use, study, modify and distribute the software to anyone and for any purpose [Cor14]. This large amount of public data makes it possible for researchers to mine project data, and various tools and datasets have been created to assist researchers in that goal [Kal+14].

GitHub

GitHub is currently the largest collaborative code hosting site built on top of the *git* version control system [Gou13; Kal+14]. It uses a "fork & pull" model, meaning developers create their copy of a repository, make changes and open a pull request when they would like the project maintainer to pull their modifications into the main branch [Kal+14]. In addition to code hosting, collaborative code review, and bug tracking, GitHub has also incorporated social features which we use as an indicator for project popularity. Its popularity, built-in social features, and availability of metadata through an accessible API have made GitHub very attractive to software development researchers [Kal+14].

GitHub API We use the GitHub REST API v3[2] to retrieve the most up-to-date information about the repositories we analyse. A primary challenge for the data collection process in this thesis was the limit of 5,000 requests per hour for authenticated requests [Gou13]. Since many repositories on GitHub did not match the properties we had anticipated to categorize them as possible study objects, this limitation forced us to use GHTorrent, an offline mirror of the GitHub API, to find suitable projects for our analysis.

The API offers a broad range of features, including the search for individual projects. This allowed us to retrieve the most up-to-date metadata for our repositories selected via GHTorrent. For this work, we primarily use the existing API client PyGitHub[3]. In some cases relevant to this work, we also extend the functionality of this client. We added custom implementations to retrieve the contributor count of a project or to determine if the project has a GitHub description or ReadMe.

GHTorrent

GHTorrent[4] offers a scalable, queriable, offline mirror of the data provided through the GitHub API [Gou13]. This data includes various information about individual GitHub

[2] `https://docs.github.com/en/rest`
[3] `https://github.com/PyGithub/PyGithub`
[4] `https://ghtorrent.org/`

projects, including contributors, programming language(s), forks, commits, and so on, but excluding actual source code [Vei16]. In this thesis, we use GHTorrent's CSV files from march[5], containing data for around 190 Million GitHub projects, to select projects for our research. Since GitHub is a dynamic site where projects are created and deleted constantly, and GHTorrent is limited in updating all data consistently [Gou13], all metadata of projects used in the study comes directly from the GitHub REST API. This prevents data from not matching the latest data at the time of project retrieval.

2.3 Static Analysis Tools

SATs analyse source or compiled code without execution to increase software quality by making developers adhere to coding standards, finding potential bug causes, or security threads [NB05]. Quality defects associated with specific code regions found by any static analyser are referred to as *findings*. While static analysers can uncover errors undetected by testing through bug pattern search, they may also produce false positives in some cases [NB05; HHS14]. Many static analysers can create findings for metric violations, such as code clones, missing interface documentation, deep nesting, lengthy methods, and long files.

This section provides basic information about commonly used SATs in Java projects, their capabilities and purposes, and how to integrate them into the development process.

Incremental Integration

Incremental quality analysis identifies the causes of defects introduced by changes, whereas non-incremental quality analysis exclusively analyzes projects in their entirety. Bauer suggests in [Bau+12] that software quality analysis should be incremental in order to provide developers quick feedback on recent changes. Therefore, development teams should use SATs that offer incremental analysis or explicitly configure their tools such that they can be used incrementally.

Integration of Static Analysis Tools

SATs can be deployed on the developer's machine or during a project's build process allowing developers to run code inspections during their development process at any time [NB05]. Developers can add their favorite static analysers via plugins to their IDEs to acquire instant feedback on their code changes. Alternatively, they can use a linters

[5] `http://ghtorrent-downloads.ewi.tudelft.nl/mysql/mysql-2021-03-06.tar.gz`

command-line interface (CLI) to run static analysis on their machine. All the linters listed below also offer CI/CD integration that allows development teams to enforce coding standards. Some of the most commonly used open-source build tools for Java projects are Ant, Gradle, Maven, and Jenkins. Tools such as Codacy[6] or Lift[7] allow the integration into code reviews (automated code reviews) where findings are added as comments to the source code, just as an actual reviewer would do. This saves reviewers time as some comments they would have to write manually are already covered by the static analyser in advance [Bal13].

Overview of Static Analysis Tools

The following section lists all SATs that support analysis for Java and are of relevance in the course of this work.

Checkstyle[8] is a highly configurable open-source single file static analyser [Che]. It allows teams to define rules to write Java code that adheres to a coding standard [Che]. It provides a CLI, allows integration via multiple build tools, and offers plugins for all major IDEs [Che]. Checkstyle can also be integrated into GitHub Pull Request (PR)s using Sticker CI[9] or Checkstyle GitHub Actions[10]. Checkstyles most significant limitation is that it only analyses the content of one file during all check executions [Che]. Therefore, it cannot determine complete inheritance hierarchies or detect clones, making it inferior to other static analysers or advanced IDEs [Che].

FindBugs[11] and SpotBugs[12] are open-source SATs that inspect Java bytecode for bug patterns occurrences [Fin]. The bug patterns are extensible; new patterns and detectors can be added through plugins like fb-contrib[13] or Find Security Bugs[14] [Spo; Fin]. SpotBugs is the spiritual successor of FindBugs, whose development stopped in 2016 [Spo]. They can be used standalone or through several integrations, including Ant, Maven, Gradle, and Eclipse [Spo].

PMD[15] is an open-source static analyser that finds common programming flaws for

[6] https://www.codacy.com/?utm_campaign=Feature_CodeScanning&utm_source=GitHub
[7] https://lift.sonatype.com/getting-started
[8] https://github.com/checkstyle/checkstyle
[9] https://github.com/stickler-ci
[10] https://github.com/nikitasavinov/checkstyle-action
[11] https://github.com/findbugsproject/findbugs
[12] https://github.com/spotbugs/spotbugs
[13] https://github.com/mebigfatguy/fb-contrib
[14] https://github.com/find-sec-bugs/find-sec-bugs
[15] https://github.com/pmd/pmd

multiple languages, focusing on Java and Apex [PMD]. PMD features built-in checks and supports an extensive API to write custom rules [PMD]. PMD is shipped with the copy-paste detector CPD; CPD uses the Karp-Rabin string matching algorithm to find code clones [PMD]. PMD can be integrated into the build process with Maven, Gradle, Ant, or its CLI, offers plugins for many IDEs,and allows automatic code review integration with Codacy [PMD].

Semgrep[16] is an open-source SAT that finds bugs and enforces coding standards for multiple languages, including Java [Sema]. It comes with many existing and the possibility to create custom rules [Sema]. It offers plugins for IntelliJ IDEA, Visual Studio Code (VSC), and Vim and can be integrated into GitHub PRs via the Semgrep CI[17] [Sema].

Infer[18] is developed by Facebook to detect bugs in Java, C, C++, and Objective-C code [Fac]. It can be integrated into the Continuous Integration (CI) with Ant, Gradle, Maven, and more. It checks for null pointer dereferences, memory leaks, coding conventions and unavailable APIs [Fac]. Infer attempts to build compositional proofs of the programs at hand by composing proofs of its constituent modules, and potential bugs are then extracted from failures of proof attempts [CD11]. Open-sourced in 2015, Infer is used at several large tech companies, including Amazon, Mozilla, and Spotify [Fac].

Moose[19] is an open-source software and data analysis platform for Java, C, C++, C#, and more [Moob]. It comes with many basic services like metrics, queries, and interactive visualizations per default [Mooc]. Written in Pharo, it allows programmers to craft custom checks cheaply and to produce more complex analyses like the computation of dependency cycles, detection of high-level design problems, or identification of exceptional entities [Mooc].

CodeQL[20] is an open-source analysis engine used by developers to automate security checks that let them query code as though it were data [Gita]. The idea of querying source code like any other type of data arose at the outset of Semmle in 2006 [Mooa]. CodeQL can extract data models from codebases for eight languages [Gita]. LGTM[21] is Semmle's corresponding code analysis platform that allows development teams to identify vulnerabilities early and prevents them from reaching production. Since 2019

[16] `https://github.com/returntocorp/semgrep`
[17] `https://github.com/returntocorp/semgrep-action`
[18] `https://github.com/facebook/infer`
[19] `https://github.com/moosetechnology/Moose`
[20] `https://github.com/github/codeql`
[21] `https://semmle.com/lgtm`

Semmle, and therefore CodeQL and LGTM are owned by GitHub. Accordingly, LGTM is free for all OSS projects and allows automatic code reviews for GitHub and Bitbucket [Semb].

SonarQube[22] is a very commonly used open-source code analysis tool in the context of CI environments [Mar+19]. SonarQube includes its own rules and configurations, custom rules can be added, and popular rules of other SATs, such as FindBugs and PMD, can be integrated [Mar+19]. Its community edition supports 15 languages, detects potential bugs and vulnerabilities, and assesses code quality metrics and quality gates [Son]. SonarQube only offers GitHub PR integration with the Developer edition, but there are open-source integrations like SonarQube GitHub Actions[23].

Coverity[24] was introduced in 2006 in collaboration with the U.S. Department of Homeland Security as the largest public-private sector research project in the world, focused on open-source software quality and security [Syn]. Coverity is now managed by Synopsys and allows OSS development teams to find and fix defects in seven languages with their community edition [Syn]. Coverity Scan provides multiple integration possibilities with GitHub [Syn].

Codacy[25] is a SAT that automates code reviews by fetching the latest modifications done in a repository for which it then produces warnings [DB15; coda]. It provides feedback for code quality and keeps track of technical debt for more than 40 programming languages [coda]. It is flexible by allowing the project owner to select from a large set of warning definitions or write custom checks [DB15]. Codacy offers a sign-up through different Git providers, including GitHub, and automatically incorporates itself into the existing Git provider workflows [codb].

Teamscale[26] is a closed-source software intelligence platform that offers static analysis for 28 programming languages. It is an incremental quality analysis tool providing feedback to developers within seconds after a commit enabling real-time software quality control [HHS14]. Through various heuristics, defects are also tracked when being moved between methods or files [SHJ14]. The reliable differentiation of new and old quality defects allows concentrating on recently introduced defects [Göd+14]. The web interface and IDE plugins make the information available for developers and other stakeholders [Göd+14].

[22] `https://github.com/SonarSource/sonarqube`
[23] `https://github.com/kitabisa/sonarqube-action`
[24] `https://scan.coverity.com/`
[25] `https://www.codacy.com/`
[26] `https://www.cqse.eu/en/teamscale/overview/`

We will use TS to analyse the quality metrics listed in Section 2.1 for the projects included in our study. TS allows the configuration of so-called *analysis profiles* to execute project-specific analysis requests.

Continuous Quality Assessment Toolkit (ConQAT)[27] is an open-source SAT that features various analysis types, targeting popular programming languages and even natural language [Vei16]. It was designed to perform research on automated software quality analysis at TUM in 2005 and was one of the cornerstones of CQSE when the company was launched in 2009 [Hum]. CQSE announced ConQATs end of life in 2018 as they view Teamscale as the superior tool [Hum].

2.4 Related Work

This thesis is not the first to investigate the impact of certain factors on code quality. Nor is it the first to investigate questions related to code quality, static code analysis, open-source projects, and GitHub. To our best knowledge, this work is the first to evaluate the integration linters in open-source projects in a large-scale study and the first to use static code analysis to analyse the effect of linters on code quality metrics in open-source projects. Due to the large number of papers covering similar topics, the following selection lists work that has significantly impacted this work and might therefore be of interest to the reader.

Programming Languages & Code Quality

This thesis greatly builds up on the master thesis [Vei16] written by my advisor Daniel Veihelmann. In his thesis, he conducted a large-scale study using open-source systems from GitHub to evaluate the impact of different programming languages on code quality. Using the SAT ConQAT a predecessor of TS, Veihelmann examined, as expected, that there are software quality metric differences among different programming languages. He found that C projects tend to contain excessively long methods significantly more often than non-C projects. We do not compare quality between projects using different languages but focus on a single language, namely Java. In addition, Veihelmann analysed potential correlations between code quality metrics and team size, project size, and popularity. He could not measure significant correlations between team size and quality, neither between popularity and quality nor between project size and quality. Besides not being able to show a clear correlation between project size and overall quality, he could identify correlations between individual metrics and project size. He,

[27] `http://www.conqat.org/`

for example, found tendencies where larger projects seem to have better interface documentation while they have more clones, methods with deep nesting, long methods, and large files. After evaluating his results, one of his assumptions is that quality assurance practices are not a standard on GitHub, especially when using the (fork, pull request, merge) cycle in team projects.

Usage of Static Analysis Tools

In 2020 Nguyen [NWA20] executed a user-centered study of developer needs and motivations regarding the use of SATs. He surveyed 81 developers in a company using a closed-source SAT from Checkmarx[28]. She found out that IDE integrations and dedicated tools are the most frequently used and that developers like to have all their warnings aggregated in a central interface. Developers in her study rank the importance of warnings related to coding style as less critical than warnings related to performance, memory, and concurrency bugs. She also found out that Developers mostly use analysis tools in their spare time and like fixing all warnings in one, preferably short working session. According to her study, Developers tend to choose warnings that they know they can fix, typically through their knowledge of the code base, their experience of the tool, and warning types. Also, developers like if the UI of an SAT encourages good behavior such as collaboration between developers and building a knowledge database.

In a similar study, [Joh+13] Johnson surveyed 20 developers using various SATs, including Checkstyle, FindBugs, PMD, and Coverity. One reason she found for developers to use SATs was availability in the development environment and most modern IDEs come with built-in static analysers. She also found out that developers are seeing negative impacts in poorly presented output as well as problems with SATs being known for producing many false positives. Customizability also seems to be of importance, and that the configuration of the tools plays a large part in the output you get. Participants complained about the complexity of the SATs' configurations, hindering them from adapting them to their individual needs and preferences.

[28] `https://www.checkmarx.com/`

3 Case Study Design

In this study, we apply a quantitative research approach [CC17] using GitHub repositories. This chapter describes the methodology and execution of the study. In particular, we will explain how we select open-source projects for our research, how the integration of linters in projects is determined, and how the project's code quality is analysed.

3.1 Overview

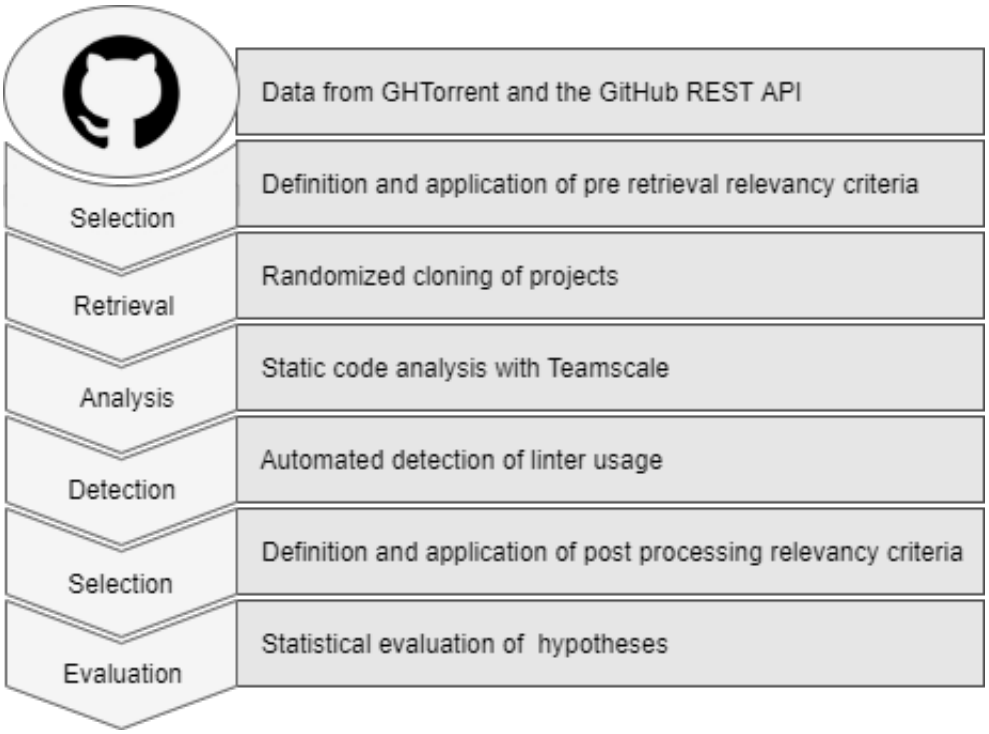


Figure 3.1: Overview of data retrieval, selection, and processing

Figure 3.1 outlines the phases in which data from GitHub is selected, retrieved, and processed to fit our criteria required for evaluation required. The *initial selection phase* uses the GHTorrent data set and the GitHub REST API to create a set of projects potentially relevant to the study. Presumably, interesting metadata of these projects is saved to a database to be available for evaluation purposes later. In the *retrieval phase*, projects from our database are cloned randomly for further processing. After retrieval, in the *quality analysis phase*, projects are analysed by the static analysis tool

Teamscale and results saved to the database. In the *linter detection phase*, we use the LinterDetectionTool (Section 3.5) to determine if a project uses a linter or not. We save the findings[1] created by the LinterDetectionTool to a database. In the *post-analysis selection phase*, we apply a second set of selection criteria to finalize the set of projects included in our study. Finally, we evaluate the collected data with statistical methods and tools to answer the research questions in the *evaluation phase*.

3.2 Research Questions

This research study's objective is to investigate the benefit the use of a linter has on the code quality in OSS systems. A quantitative research approach using public GitHub repositories is applied [CC17]. Before conducting the study, the research questions are formulated and from them several directional hypotheses are developed, which represent the expected results [CC17]. All research questions and hypotheses were written before the execution of the study, as this is considered best practice [Con15].

In the course of this work, we will address the following research questions and hypotheses:

RQ 1: *What distribution do linters have among open-source projects?*

This question aims at investigating the occurrence of linters in open-source projects. As we examine the impact of linters on code quality, the frequency in which linters are used could be an implicit indicator of their impact on project's code quality. Especially a high occurrence among team projects would potentially show how relevant the use of linters appears in development teams. *H 1* predicts that linters are more common among larger software projects, as we assume that integrating a linter into a project happens at a certain point when projects reach a specific size at which the overall comprehensibility for a single person becomes impossible. Large projects are expected to use linters as a quality assurance for new code added to the project, improving the quality of the traditional code review [Bal13]. *H 2* inspects the impact of the team size on the occurrence of linters in projects. We expect that projects with larger teams have a higher urge to assure quality through automated reviews with linters. According to Bosu, PRs by *core* developers[2] are reviewed and accepted faster

[1] The LinterDetectionTool itself is a linter that searches the content of a project for signals that indicate linter usage and creates a *finding* that contains data about the location, the detected linter, and the causing signal.

[2] A developer who consistently contributes to a project and knows the project well

and more often than by *peripheral* developers [3] [BC14]. Linters can potentially reduce this gap by enforcing coding standards and speeding up the review process. As linters have different analysis procedures and produce varying findings, teams concerned about code quality might utilise more than just one linter. We would, for instance, expect that projects using Checkstyle to enforce coding standards additionally use SpotBugs to find bugs. In *H 3*, we, therefore, assume that projects are likely to use multiple linters.

RQ 2: *Do projects using linters have better code quality than projects that do not use a linter?*

This question aims to answer the paper's problem statement. Therefore we compare the distributions of the quality metrics clone coverage (*H 4*), comment completeness (*H 5*), nesting depth violations (*H 6*), method length violations (*H 7*), and file size violations (*H 8*) between projects using and not using linters. As different linters create different warnings, we consider different sets of linters in each of the hypotheses. Checkstyle, for example, is a single file static analyser and is incapable of detecting code clones. Therefore projects using Checkstyle will be excluded from the evaluation of *H 4*. The exact set of linters used for each hypothesis are specified in the respective sections below. The hypotheses listed in Table 3.1 are all assuming that projects with linter integration have better metric distributions.

RQ3: *Are projects using linters more popular than projects that do not use a linter?*

As already mentioned repeatedly, there is a clear correlation between the success of a software system and the software's quality. *RQ 2* aims to further explore this relationship by analyzing whether there is a dependency between linters usage and OSS success. Inspired by the like button of modern social networks like Facebook [BV18], GitHub users can also star a repository, presumably manifest interest or satisfaction with the hosted project [BBS13]. Veihelmann assumes that developers on GitHub might give stars to GitHub projects whose code quality they appreciate [Vei16]. This claim is supported by the existing correlation between the number of forks and the star count of GitHub projects [Bor+15]. Despite Veihelmanns conclusion that a project's code quality does not correlate with its popularity on GitHub [Vei16], we will investigate if there is a correlation between using a linter and the popularity of a project on GitHub. *H 9*, therefore, assumes that projects using linters have more GitHub stars than other projects. In other words, they are more popular.

[3] A developer who occasionally or rarely contributes to a project

3.3 Hypotheses

This section will go into more detail about our hypothesis and present our assumptions and evaluation approaches.

Hypotheses	
H 1	Linters are more frequently used in large compared to small software projects.
H 2	Linters are more frequently used in software projects with bigger development teams.
H 3	Projects that use at least one linter are likely to utilise other linters also.
H 4	Projects not using linters have more code clones than projects using linters that support code clone detection.
H 5	Projects not using linters have less interface documentation compared to projects using linters that support findings for missing interface documentation.
H 6	Projects not using linters have inferior nesting-depth assessments than projects using linters that support findings for nesting depth violations.
H 7	Projects not using linters have higher percentages of methods exceeding a critical length than projects using linters that support findings for method length violations.
H 8	Projects not using linters have higher percentages of files exceeding a critical length than projects using linters that support findings for file size violations.
H 9	Projects not using linters have less stars than projects using linters.

Table 3.1: Overview of hypotheses examined in this study

Hypothesis 1 (H 1) *Linters are more frequently used in large compared to small software projects.*

Linters aim to improve code quality and can help developers adhere to coding standards. Small projects have less code and are therefore easier to maintain. As a system's size increases, so does maintenance effort [Bal13]. Taking this into account, we hypothesize that larger projects have a higher need to utilize linters to resolve potential defects early.

Evaluation To validate this hypothesis, we will group projects by size (SLOC) and compare the percentages to which our LinterDetectionTool detects liners in the groups.

We will split our collected data into three groups. We consider projects ranging from 1 to 10 thousand SLOC (KSLOC) are considered **small**, from 10 to 100 KSLOC **medium**, and above 100 KSLOC **large**. If our assumption is valid, we should see an evident increase in relative linter detection moving from small to large. If we do not observe an increase of at least 5%, we reject our assumption.

Hypothesis 2 (H 2) *Linters are more frequently used in software projects with bigger development teams.*

Similar to *H 1*, we additionally want to investigate the impact of team size on linter usage. The more developers work on a project, the less the individual knows about the project's entirety. Besides that, over 80% of OSS developers are peripheral developers [BC14]. The bigger the team, the more likely it is that we have a larger number of peripheral developers than core developers. Therefore, those core developers have to review more code, so they possibly introduce SATs to reduce their review effort. This introduction allows them to make peripheral developers adhere to coding standards and avoid easy to detect defects early and automatically.

Evaluation To validate this hypothesis we will first group projects by size (SLOC), according to the subdivision of H 1, and then by the number of contributors. We do this to eliminate the potential effect of the projects' sizes on team size. For better visualization, we will create a bar chart for each of the size groups mentioned above. The bar chart will represent the relative linter detection in correlation to team size, starting from smaller team sizes (left) to bigger team sizes (right). If our assumption is true, we will have more or less constantly increasing regression lines. If our regression lines are decreasing, we reject our hypothesis.

Hypothesis 3 (H 3) *Projects that use at least one linter are likely to utilise other linters also.*

This hypothesis is based on the idea that individual linters perform different checks and have varying benefits. While, for example, Checkstyle assessments aim to find coding style violations by checking source code, FindBugs seeks to find bugs by looking for bug patterns in the byte code. Therefore a team can possibly benefit from using multiple different linters trying to find the maximum amount of potential *code smell*[4]. More advanced analyzers like Teamscale or Sonarqube already allow integrating findings of other linters via report upload or have a direct integration that allows executing other linters checks.

Evaluation To validate this hypothesis, we will count the detected linters for each

[4] Any characteristic in the source code of a program that possibly indicates a problem

project for which the LinterDetectionTool (Section 3.5) created the required amount of findings. We will display the distribution in a bar chart diagram. Our assumption holds if a relatively large amount of projects that use one linter also use a second linter. We define our threshold to be 50%. We view this as a high percentage, as some tools like SonarQube create warnings for code quality and security. Therefore, the integration of additional linters might not be necessary. From our manual evaluation for the LinterDetectionTool (Section 3.5), we, on the other hand, do expect Checkstyle and FindBugs to be the most occurring linters. And when using a tool that only focuses on style violations, like Checkstyle, a need for an additional linter that searches for bugs, like SpotBugs, FindBugs, or PMD, might arise. If over 50% of projects use only exactly one linter, we reject our assumption.

Evaluation approach for *H 4* to *H 8*

In order to evaluate our hypotheses *H 4* to *H 8*, we need specific criteria to judge the validity of a particular assumption. For each hypothesis, we will potentially have multiple comparisons between different groups. Each comparison will be between two disjoint sets of projects, l_1 and l_2. Our goal is to examine if a metric in set l_1 is better than in l_2. We use NumPy[5], SciPy[6], and Matplotlib[7] for all statistical tests mentioned in the subsequent parts. The characteristics of the comparisons and the methodology we use to evaluate them are based of on Veihelmann's work [Vei16].

Wilcoxon Tests for Group Similarity

To investigate whether one value distribution is significantly similar to another, we apply a *Wilcoxon signed-rank test*[8] (short: Wilcoxon test) [Fah+16]. This non-parametric test can be used for two independent groups whose distributions have the same shape but may be shifted by an amount [Fah+16]. We take this as given for our metric value distributions.

The primary outcome of the Wilcoxon test is a probability value (p-value) that denotes the likelihood that our two groups originate from the same probability distribution [Fah+16]. The p-value indicates whether we may see the distributions as significantly different or rather similar [Fah+16]. We adopt the p-value of $p = 0.01$ from Veihelmann in [Vei16]. The *significance level* or *alpha* of 0.01 describes the probability of assuming our two sets are different even though they are not. We conclude that two metric distributions are significantly different if the resulting p-value falls below our defined threshold value of

[5] https://numpy.org/
[6] https://www.scipy.org/
[7] https://matplotlib.org/
[8] more popularly known as *Mann–Whitney U* test

0.01. If $p < 0.01$ holds between the two sets, we use a comparison of the medians as well as a visual interpretation of the corresponding boxplots to conclude which of the two sets has a superior metric distribution.

Hypothesis 4 (H 4) *Projects not using linters have more code clones than projects using linters that support code clone detection.*

Some linters can detect code duplicates of type-2. Therefore we expect projects using these linters to have fewer code clones of type-2 than other projects. The linters PMD (CPD) and SonarQube can detect clones of type-2 [PMD; KAY14]. Also, we consider that small projects are likely to have fewer code clones than medium or large projects [Vei16].

Evaluation In order to validate this hypothesis, we compare the corresponding clone coverage distributions while taking the usage of linters mentioned above into account. Therefore we will compare a set of projects using the four linters mentioned above to a group of projects that do not use any linters. To avoid that the outcome is affected by a more significant number of small projects in the set of projects not using linters, which is expected using random selection and assuming *H 1* to be accurate, we make different comparisons for the size groups small, medium, and large. A Wilcoxon test for the two sets is applied to evaluate whether the respective value distributions are significantly different. Accordingly, we accept our assumption that projects not using linters are more prone to code duplication as confirmed if the clone coverage values of repositories with the four linters, supported by Wilcoxon tests, are significantly lower than of projects without linter in at least two of the three groups. Otherwise, we will reject our hypothesis.

Hypothesis 5 (H 5) *Projects not using linters have less interface documentation compared to projects using linters that support findings for missing interface documentation.*

This hypothesis is based on the idea that some linters create warnings for missing interface documentation. Linters that generate findings for missing Javadoc Comments are Checkstyle, SonarQube, PMD, Codacy, and Coverity. Therefore we expect projects using these linters to have better interface documentation than projects that do not use any linters.

Evaluation To validate this hypothesis, we will compare the comment completeness ratios between a set of projects using the linters mentioned above against projects not using linters. Taking Veihelmanns evaluation of comment completeness in small versus large projects in [Vei16] into account, which showed that small projects tend to have worse comment completeness, we make separate comparisons for small,

medium, and large projects. A Wilcoxon test for each pair of sets in comparison is applied to evaluate whether the respective value distributions are significantly different. Accordingly, we accept our assumption if the comment completeness values of repositories with mentioned linters, supported by Wilcoxon tests, are significantly higher than those of projects without linters for the majority of our comparisons. Otherwise, we will reject our hypothesis.

Hypothesis 6 (H 6) *Projects not using linters have inferior nesting-depth assessments than projects using linters that support findings for nesting depth violations.*

The linters listed in *H 5* are also capable of creating warnings for nesting depth violations. Accordingly, we expect projects using these linters to have fewer nesting depth violations than projects that do not use any linters.

Evaluation The evaluation approach is identical to the one applied in *H 5*. Additionally, we will differentiate between our yellow and red percentage assessments. Therefore we will have a total of six comparisons, two comparisons for each of our three pairs of two sets of small, medium, and large projects. If the majority of comparisons are in favor of our assumption, we confirm that projects not using linters have inferior nesting-depth assessments than projects using linters that support findings for nesting depth violations. Otherwise, we will reject our hypothesis.

Hypothesis 7 (H 7) *Projects not using linters have higher percentages of methods exceeding a critical length than projects using linters that support findings for method length violations.*

The linters listed in *H 5* are also capable of producing findings for method length violations. Consequently, we expect projects using these linters to have fewer method length violations than projects that do not use any linter.

Evaluation Since our method length metric is also an assessment metric, the evaluation approach is identical to *H 6*.

Hypothesis 8 (H 8) *Projects not using linters have higher percentages of files exceeding a critical length than projects using linters that support findings for file size violations.*

The linters listed in *H 5* are also capable of producing findings for file size violations. Therefore we expect projects using these linters to have fewer file size violations than projects that do not use any linters.

Evaluation Since the file size metric is an assessment metric, the evaluation approach is identical to *H 6*.

Hypothesis 9 (H 9) *Projects not using linters have less stars than projects using linters.*

This hypothesis aims to answer *RQ 3*; are projects using linters more successful than projects which do not? The GitHub star option allows users to show that they like a repository. We use these stars to measure the popularity and, therefore, the success of our study objects.

Evaluation

To validate this hypothesis, we will compare the amount of GitHub stars in projects using and not using linters. Since there might be a correlation between project size and popularity, we again divide our research objects into small, medium, and large projects to make individual comparisons like described in the evaluation approach of *H 4*.

3.4 Study Objects

This section aims to describe the attributes qualifying objects for our work. As we strive to compare OSS projects from GitHub with each other, we urge to compare projects of similar size and development history. Projects are randomly selected and analyzed to gain insights into their code quality, but the comparison only makes sense if they fulfill the required criteria. This section provides an overview of the aspects necessary for a project to qualify for the evaluation phase of this study. Some inclusion requirements will be available before projects are downloaded and some other after analysis. Most of the selection requirements are based on Veihelmanns master thesis [Vei16].

Project Selection Criteria

Pre Download Requirements

We define a set of criteria each software project must meet to qualify for download and analysis. They are defined as follows:

1. The project must be available on GitHub.
2. The project must not be a fork or mirror of another project.
3. The project must include a ReadMe or have a GitHub description.
4. The project must have between 100,000 bytes and 214 MB of Java code.

The following paragraphs show the reasoning behind the chosen criteria:

ad 1 (Available on GitHub)
The most obvious point is the availability on GitHub, as GitHub is our only data source. GHTorrent might include deleted repositories, but as all data included in our study comes from the GitHub REST API, deleted projects are excluded.

ad 2 (No fork or mirror)
A fork is a copy of a repository and is most commonly used to propose changes to someone else's project or to use someone else's project as a starting point for a new project [Gitb]. To duplicate a repository without forking it, a mirror of the repository can be created [Gitb]. The mirror is an exact copy that stays in sync with its parent project [Vei16]. As Veihelmann observed, including forks and mirrors could lead to the incorporation of almost identical code being taken into account during statistical evaluation [Vei16].

ad 3 (ReadMe or description required)
We see two reasons for the requirement of a description or ReadMe file. First, as Veihelmann observed, it is hard and time-consuming to assess a repositories' purpose when they have don't provide basic information about themselves [Vei16]. This requirement makes manual analysis of projects more manageable. Also, manual analysis of projects for the development of the LinterDetectionTool showed that linter integration is often displayed or announced in the ReadMe file of a project. Lastly, a missing ReadMe or description could indicate that the project is not developed for public use, and therefore quality factors might be of low importance.

ad 4 (Quantity of source code)
Veihelmann states that 18,000 bytes of code represent approximately 1,000 LOC [Vei16]. Since we are not interested in particularly small projects, we picked 100,000 bytes of Java code as the lower bound. As upper bound, we use the 214 MB of Java code as proposed by Veihelmann in [Vei16]. He observed that unusually high volumes of source code could likely be due to multiple independent software packages within one individual repository or the master branch containing older versions of the project, making them unsuitable for our study [Vei16].

Post Analysis Exclusion Criteria

To ensure data relevancy and integrity after download and analysis we define a set of exclusion criteria. Projects meeting these are excluded from our study. These criteria are based on data measurements that are not available to us prior to our analysis by the LinterDetectionTool and or static code analysis with Teamscale. The post-analysis exclusion criteria are as follows:

1. The repository has less than 1000 SLOC.
2. The determined clone coverage of a project is higher than 75%.
3. The LinterDetectionTool generates an ambiguous result.

Detailed explanations for the points stated above can be found in the following paragraphs:

ad 1 (Quantity of source code)
Veihelmann has shown that some quality metrics like method length and file size are affected by the size of a project [Vei16]. Projects with less than 1000 SLOC, after code exclusions have been performed, might therefore cause faulty analysis results when

comparing small projects. For example, a project with less than 500 SLOC can never have a single red file size violation and will always result in an r_p value of 0%.

ad 2 (Clone coverage)
Veihelmann identifies two causes for very high clone coverage, namely the inclusion of multiple versions of (almost) identical software projects and sub-projects that appear to be independent but were created by copying an original project numerous times, including only minor adaptations [Vei16]. High clone coverage numbers, therefore, often don't represent the numbers of a single software project. Thus we aim to exclude such repositories. Veihelmann conducted a manual evaluation to identify a threshold at which projects are likely not to contain a single software project [Vei16]. He found that 14 of 30 manually analysed projects with a clone coverage between 75% and 80% contained multiple (versions) of programs following the description above [Vei16]. Assuming that this trend is also valid for even higher amounts of cloning, he chooses a clone coverage value of 75% as the upper limit. We adopt this value for our work and exclude projects with clone coverage above 75%.

ad 3 (Ambiguous LinterDetectionTool results)
From our manual evaluation of the LinterDetectionTool (Section 3.5), we have elaborated that there are projects where we are uncertain whether or not they are using an SAT. Since we want to distinguish between projects using and not using linters clearly, we exclude projects where we are unsure about linter integration. There additionally exists a set of linters that is excluded from the study for various reasons. Projects potentially using any of these linters are also excluded from the study. The exclusion of ambiguous LinterDetectionTool results is elaborated further in Section 3.5 *LinterDetectionTool*.

Project Selection and Criteria Application Process

The actual selection of projects consists of three steps. In the first step, we utilize GHTorrent to identify a set of possibly suitable projects. We acquire the most recent data for the above-identified projects via the GitHub REST API in step two. Only if the data fulfills the pre-download requirements, the project is cloned and analysed. Lastly, after analysis, we only select projects for our evaluation that satisfy our post-analysis requirements.

Project Selection Step 1: GHTorrent

To find projects that would potentially fit our study requirements, we made use of GHTorrent, from which we downloaded the MySQL dump from 03/06/2021. The database

dump contains information on around 190 Million Projects. To find projects that are likely to match our pre-download requirements, we only select projects with at least 54,000 bytes of Java code (around 3,000 LOC). We identified a total of 606,010 projects that we could potentially include in our study if they met our pre-download requirements. We saved the URLs for these projects to find the projects' latest data via the GitHub REST API.

Project Selection Step 2: GitHub REST API

We randomly selected URLs from our acquired set of 606,010 GitHub URLs. If the project exists on GitHub, we take the data provided by the GitHub REST API to determine if the project fulfills our pre-download requirements. If the project meets the requirements, we download and analyse it using TS and the LinterDetectionTool. We save our analysis results and data from the GitHub REST API to a database for each project. This database contains the code quality analysis results produced by TS (Section 3.6), the findings created by the LinterDetectionTool (Section 3.5), and data from the GitHub REST API, including the project's URL, the number of stars, forks, commits, and contributors.

Project Selection Step 3: Post Analysis Exclusion Criteria Application

Having all necessary data for our study available in a database allows us to select only projects from the database that satisfy our post-analysis exclusion criteria for our evaluation. The SLOC of a project and the clone coverage are available from our TS analysis. Having all the findings produced by the LinterDetectionTool available allows us to count the findings produced for each linter. We consider a linter to be included in a project if the number of findings detected by the LinterDetectionTool exceeds the manually elaborated threshold in Table 3.4. If the LinterDetectionTool doesn't create findings for a project, we consider it a project that does not integrate a linter.

As we will see later, it is not feasible to accurately tell that a project does not use a linter; we cannot even prove it via manual inspection. However, our research questions and hypotheses aim to prove that projects that use SATs have better code quality than those that do not. So if we can confirm the hypotheses, we assume that without false negatives the significance would be greater or equal.

Projects with LinterDetectionTool findings that never exceed the threshold for any linter are considered projects with ambiguous linter use, and we exclude them from the evaluation. The way the LinterDetectionTool works is elaborated further in section 3.5 *LinterDetectionTool*.

3.5 LinterDetectionTool

To automatically detect the usage of linters in projects, we implemented a LinterDetectionTool[9]. The following section describes the difficulties in detecting linters, the tool's implementation process, and the final result.

Development Approach

There is no single unique indicator that allows a clear, indisputable assessment of the usage of a linter within a project. Therefore one can not automatically tell with one hundred percent certainty if a linter is in use in the development process of a project. Hence the tool does not directly state if a linter is being used but generates *findings* indicating the utilization of a certain linter within a project. We identified a large set of static analysis tools for Java (Table 3.3) that are potentially relevant to our study and the different ways to integrate them into a project's development process. Developers are potentially using IDEs or linters individually; we neglect these cases as we are only concerned with linters integrated into the development process of the entire project.

We will inspect the project's files and file content to detect potential *signals* for linter usage automatically. We try to integrate the linters from Table 3.3 into our tool by searching their documentations to find potential signals indicating integration. For each linter, we specify a list of regular expressions (*signals*) for file names and contents that could indicate the usage of a linter. The final set of signals used for the study's analysis procedure is listed in the Appendix under *Linter Detection Analysis Details*. The tool automatically searches projects for signals and, if a signal is found, it creates a *finding* (Table 3.2) with the signal's location and the linter it indicates.

In addition, we create a blacklist with signals that might indicate the usage of a linter not included in our study. If we find a blacklisted signal in a project, it can never be considered as a project that does not use a linter. The blacklist aims to minimize the likelihood that a project with zero findings uses a linter.

Finding Type	Linter	Signal	Location	Line
File Name	Checkstyle	google_checks.xml	.../google_checks.xml	-
File Content	PMD	maven-pmd-plugin	pom.xml	137

Table 3.2: Examples for findings produced by the LinterDetectionTool

[9] https://github.com/KorbiQWeidinger/LinterDetectionTool/

Name	Language(s)	Clone Detection	Coding-Style	Bug/ Security	Stars	Forks
Infer	4 Java	No	No	Yes	12.3k	1.7k
Checkstyle	Java	No	Yes	No	6k	7.8k
SonarQube	15	Yes	Yes	Yes	5.8k	1.5k
PMD	8	Yes	Yes	Yes	3.4k	1.2k
Semgrep	19	No	Yes	Yes	4k	168
CodeQL	8	No	Yes	Yes	2.8k	589
SpotBugs	Java	No	No	Yes	2.3k	380
FindBugs	Java	No	No	Yes	656	162
Moose	7	Yes	Yes	Yes	97	30

Table 3.3: Linters integrated into the LinterDetectionTool before manual analysis sorted by stars and forks on GitHub.

Manual Evaluation Approach

To test the effectiveness of the detection mechanism, we first randomly select and analyze Java projects from GitHub with the LinterDetectionTool. Afterwards, we manually evaluate the findings the tool created for the selected projects. As previously mentioned, our starting point consists of signals we created for the linters that we expect to be popular (Table 3.3). The evaluation aims to remove or adapt signals that produce false positives to improve the tool's accuracy. We also aim to investigate findings caused by signals on the blacklist to remove false positives. We will potentially add linters we did not expect to be relevant if the blacklist causes a significant amount of true positives for them. The blacklist includes signals for Codacy, Coverity, DeepSource[10], Veracode[11], and more.

To further increase the accuracy, we aim to define a required number of findings (finding threshold) for each linter. The threshold correlates with the number of findings necessary for us to consider a linters integration as *very* likely. We will manually group the projects with findings into projects with and without linter integration for each linter. We then compare the number of findings in the two groups to establish a lower limit threshold under which linters are unlikely to be used, based on our manual assessment. The manual evaluation results will be displayed as box plot diagrams showing the distribution of the absolute findings for a linter and the distribution in the two groups.

[10] `https://deepsource.io/`
[11] `https://veracode.github.io/`

Manual Evaluation Results

We randomly selected 750 existing projects from our collected GHTorrent projects (Section 3.4) and automatically analysed them using the LinterDetectionTool, saving the findings to text documents. There was a total of 163 projects without findings, which we would therefore consider as projects not using a linter. The LinterDetectionTool accordingly created findings for 587 projects, which we examined manually for a large part.

In summary, the detection mechanism instantly worked well for the linters Checkstyle, FindBugs, SpotBugs, and SonarQube. The signals for the linters PMD and CodeQL needed adaptation. There were zero findings for the signals we created for Semgrep and 5 false positives for Moose; therefore, we moved their signals to the blacklist. Infer created false positives for 427 projects and not a single true positive; for this reason, we also had to move Infer to the blacklist. Therefore we do not consider the linters Semgrep, Moose, and Infer in our study.

During the manual analysis, we discovered that repositories with an extremely high number in findings compared to other projects are likely the repositories for the respective linters themselves or additions to the linters. For example, the SonarQube repository caused 307 findings for SonarQube or the PMD repository 63 findings for PMD. Therefore, we decided to introduce upper limits of 100 or 50 in our boxplot comparisons to make them more concise. The box plots contain the number of findings in relation to our manual categorization of linter use.

In the following paragraphs, we will further share the manual evaluation results of the individual linters in more detail.

Checkstyle: There were findings for Checkstyle in 216 of the 750 analysed projects. We manually evaluated 50 of the projects randomly. It became clear that most projects that produced a finding for Checkstyle are using Checkstyle. We only find three false positives among the 50 projects we manually investigate. When there are only 1 or 2 findings for Checkstyle in a project, and it is not a false positive, it often uses a *PREUPLOAD.cfg* file that configures Checkstyle for the project. Projects that produce a higher number of findings often integrate Checkstyle using Maven or Gradle. As a result of our manual analysis (Figure 3.2), we decide that a project with more than two findings for Checkstyle is very likely to be using the linter and therefore included in our study as projects using Checkstyle. Therefore we do not consider projects with one or two findings for Checkstyle as projects using Checkstyle.

FindBugs: A total of 190 projects caused the LinterDetectionTool to create findings for

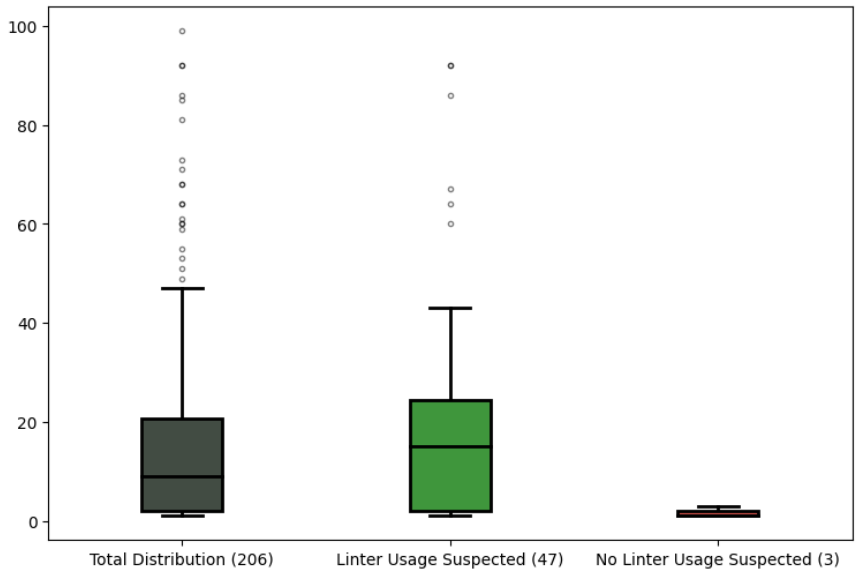


Figure 3.2: Checkstyle Manual Analysis Result (excluding outliers of over 100 findings)

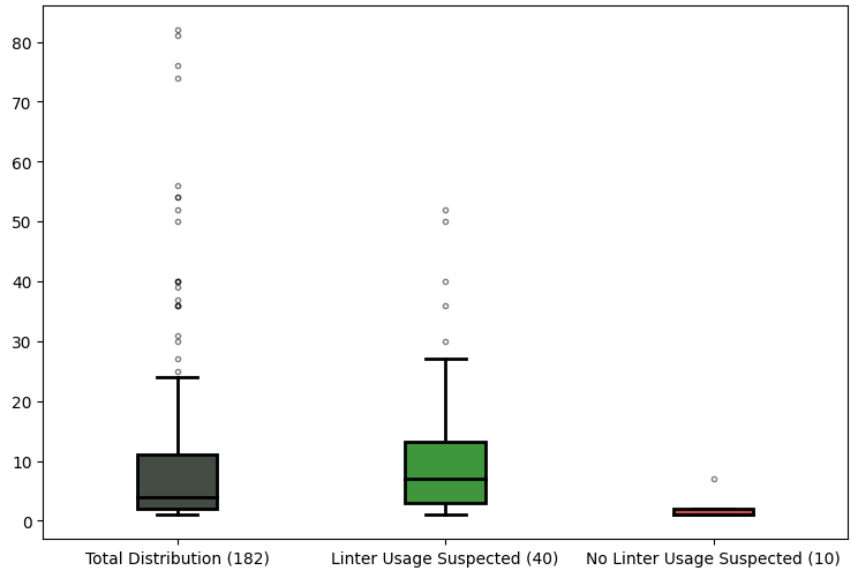


Figure 3.3: FindBugs Manual Analysis Result (excluding outliers of over 100 findings)

FindBugs. We randomly selected 50 for a manual evaluation. It became clear that the detection is very accurate and that only a view false positives were created. Projects with three or fewer findings often didn't integrate FindBugs, and the findings were caused by unrelated comments or similar. It also became clear that projects that had already transitioned to SpotBugs often still caused findings for FindBugs. We do not consider this a problem since the linters FindBugs and SpotBugs are not decisive factors in our evaluation. As a result of our manual evaluation (Figure 3.3) we decide that a project with more than three findings for FindBugs is very likely to be using the linter and therefore included in our study as a project using FindBugs. A project with three of fewer findings for FindBugs is accordingly not considered as a project using FindBugs.

SpotBugs: There was a total of 29 projects with findings for SpotBugs. We manually evaluated all of them. Some projects that integrated SpotBugs were likely to not use the linter during daily development as flags like `spotbugs.onlyAnalyze`, or `spotbugs.skip` were set to true. Arguably these projects do not use SpotBugs during daily development but might sometimes consider and resolve findings created by SpotBugs. Manually looking at the files, it became clear that the LinterDetectionTool would also create findings for FindBugs since the old FindBugs configuration was being reused to update to SpotBugs. As a result of our evaluation (Figure 3.4), we can see that projects with more than four findings for SpotBugs are likely to be using the linter and therefore included in our study as projects using SpotBugs. We do not consider projects with four or fewer findings for SpotBugs as projects having the linter integrated.

SonarQube: A total of 50 projects caused the LinterDetectionTool to create findings for SonarQube. We manually evaluated all the projects to group them into projects using and not using SonarQube. We mainly categorize projects as not certainly using SonarQube despite causing a few findings due to the signal `SonarQube` in one Java comment or excluding files created by `sonarlint` via the `.gitignore`. We do not consider a single mention of SonarQube as sufficient to tell whether the linter is being used in their development process. As a result of our manual evaluation (Figure 3.3), we decided that a project with more than three findings is likely to use the linter. Therefore projects with more than three findings for SonarQube are considered to be integrating SonarQube, while projects with three or fewer findings are categorized as projects with uncertain SonarQube integration.

PMD: The LinterDetectionTool created findings for PMD for 75 projects. We manually evaluated the findings from 50 of the 75 projects to assess our detection mechanism for PMD usage. We found the signal words `pmd`, `.pmd`, and `pmd.` to mainly cause false positives due to variable names, code comments, or `pmd` being part of randomly

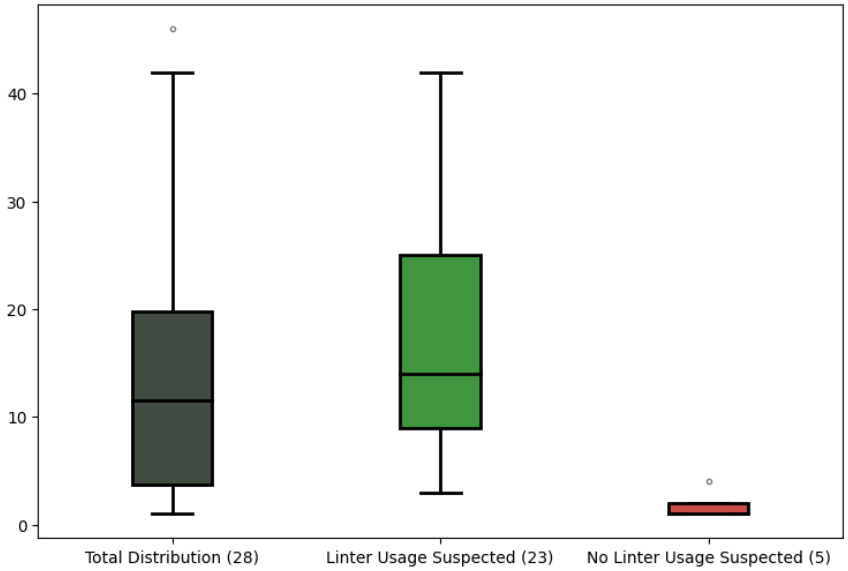


Figure 3.4: SpotBugs Manual Analysis Result (excluding outliers of over 100 findings)

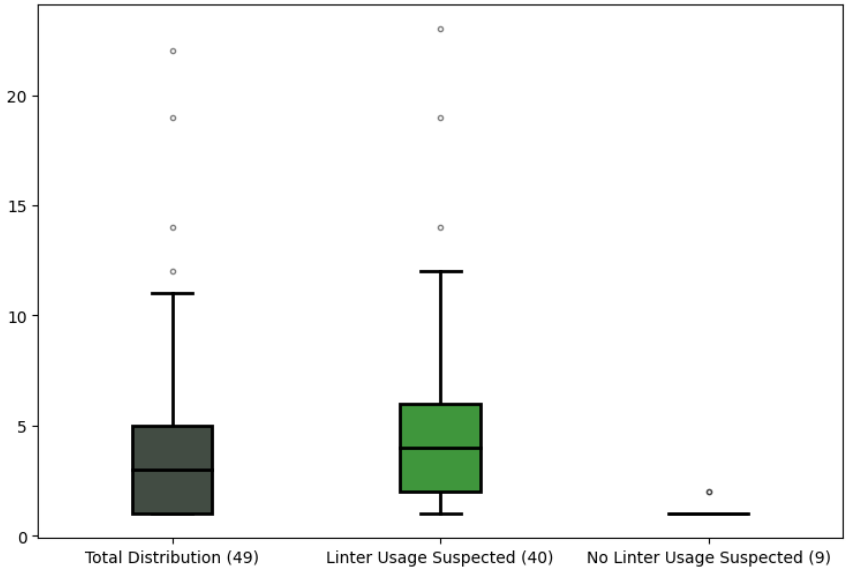


Figure 3.5: SonarQube Manual Analysis Result (excluding outliers of over 100 findings)

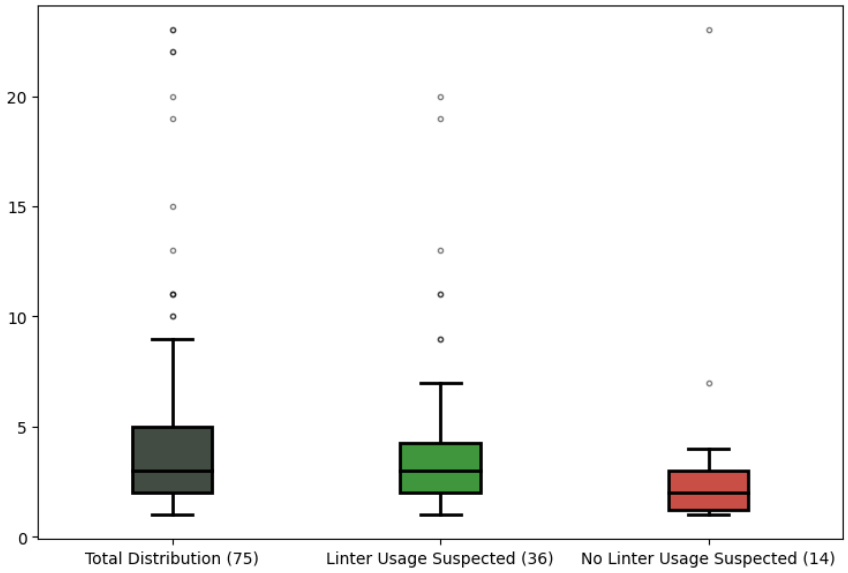


Figure 3.6: PMD Manual Analysis Result (excluding outliers of over 50 findings)

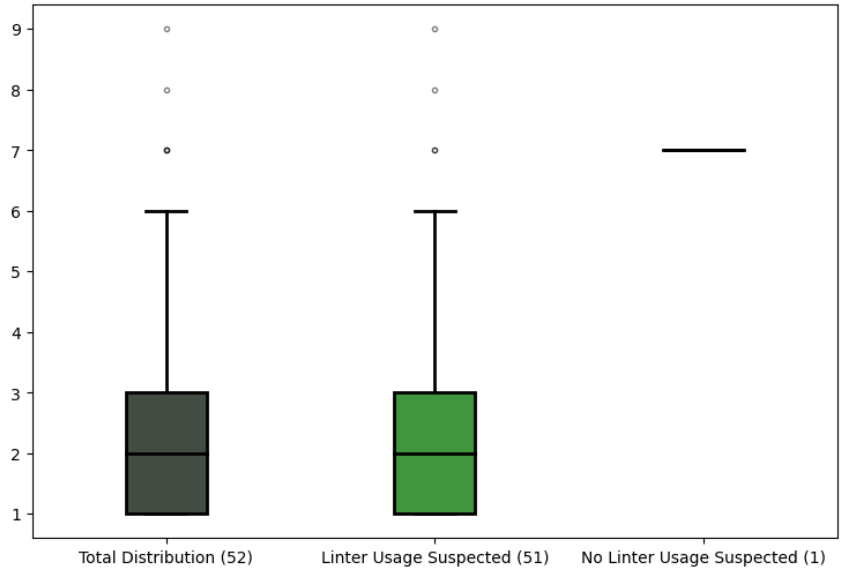


Figure 3.7: PMD Manual Analysis Result After Adaptation (excluding outliers of over 50 findings)

generated file contents. Moving these signals to the blacklist would only result in a single true positive we decided to remove them from our set of signals altogether. Our manual evaluation result before removing these signals is shown in Figure 3.6. Reevaluating the 75 projects with the updated signals now leads to 52 projects in which we suspect PMD usage, from which we manually categorize 51 as actually using PMD (Figure 3.7). The single outlier was caused by `␣PMD` being part of Java variable names; we did not remove `␣PMD` since it generated true positives in other projects. Despite only having one outlier in Figure 3.7, we aim to avoid including projects that randomly use `␣PMD` in a Java variable names or comments. Therefore we decide two findings for PMD to be our threshold, meaning we consider projects with more than one finding for PMD to be using PMD. Accordingly, we do not consider projects for which the LinterDetectionTool creates a single finding for PMD as projects using PMD in this paper.

CodeQL: There was a total of 15 projects for which the LinterDetectionTool created findings for CodeQL. Our manual evaluation resulted in 9 true positives (Figure 3.8). These projects all used the same filename `codeql-analysis.yml` for their LGTM configuration file. All false positives were caused by the signals `LGTM` and `lgtm.`, which we added to identify the usage of Semmle's corresponding code analysis platform LGTM. Therefore we removed the signals `LGTM` and `lgtm.` and reanalysed the 15 projects. This analysis rerun respectively resulted in only true positives (Figure 3.9). We choose one as a threshold for CodeQL as we did not find any of our signals to create false positives. Therefore all projects with findings for CodeQL are considered projects using CodeQL in this study.

Infer: Despite the LinterDetectionTool generating findings for Infer in 427 projects, we could not identify a single project using Infer. Since *infer* is an English word and part of other English words like **infer**ior, **infer**ence, or mis**infer**ring signals like `infer` or `Infer` caused many false positives. Therefore we removed these signals. After removal, we reanalysed the 427 projects, which resulted in zero projects for which the LinterDetectionTool generated findings for Infer. Therefore we moved the remaining Infer signals like `inferPlugin` or `com.facebook.infer` to the blacklist. Thus we do not, despite its popularity on GitHub, consider the SAT Infer in this study.

Blacklist: The LinterDetectionTool created findings due to signals in the blacklist for 309 projects. Most of them were findings caused by the signal `.*check.*.xml` which either caused a false positive or a true positive for Checkstyle; hence we removed it from the signal set of the LinterDetectionTool. The signal `jtest` for the SAT Parasoft Jtest[12] only

[12] `https://www.parasoft.com/products/parasoft-jtest/java-static-analysis/`

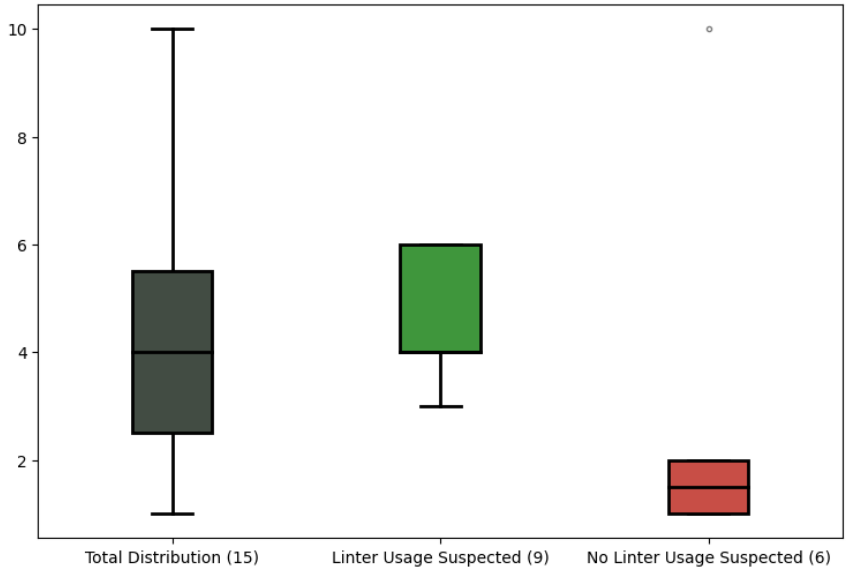


Figure 3.8: CodeQL Manual Analysis Result

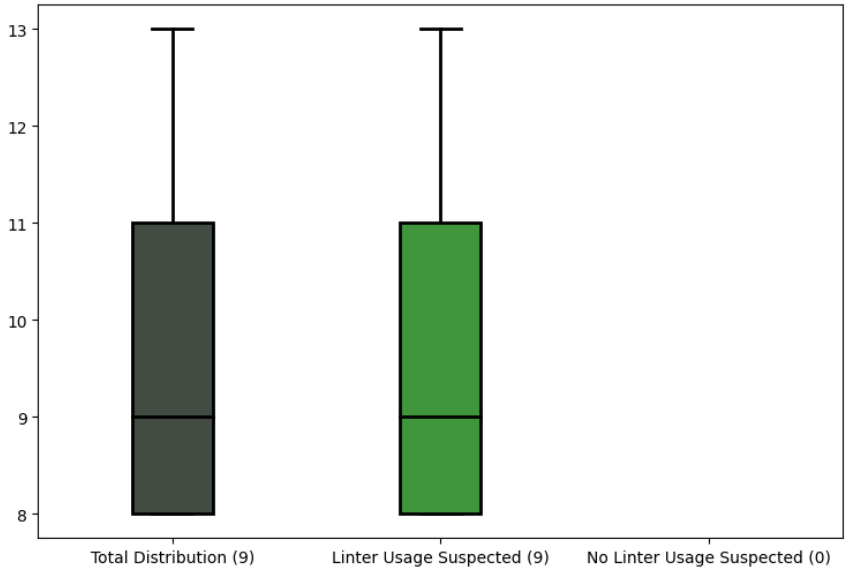


Figure 3.9: CodeQL Manual Analysis Result After Adaptation

caused false positives due to SLF4J[13], a commonly used logger; hence we also removed it. We included the signal `fortify` to detect the static analyser Fortify[14]. The signal `fortify` created some true positives but mainly false positives due to an encryption service[15] with identical name. The signals we included for Coverity and Codacy created true positives for multiple projects and zero false positives. Therefore we decided to integrate Coverity and Codacy into our study. Since we did not find any false positives for the signals signaling the usage of Coverity and Codacy, we chose their thresholds to be one.

Whitelist

We observed many false positives caused by the LinterDetectionTool within files with endings like `.txt`, uncommon endings, or without ending. We also observed that if such files generated true positives, these projects generally also had a great amount of true positives within files with unambiguous file endings like `.xml`, `.yml`, or `.gradle`. We, therefore, added a whitelist for file names to the LinterDetectionTool to avoid a large number of false positives. The LinterDetectionTool will only search file content for signals if the filename contains one of the regular expressions listed in the whitelist. The whitelist can be found in the Appendix under *RegExp Whitelist for File Names*.

Second Manual Evaluation

The first manual evaluation introduced significant changes to the signals included in the LinterDetectionTool and changed the set of linters we considered for the evaluation part of this study before. The linters included in our research now are Checkstyle, Codacy, CodeQL, Coverity, FindBugs, PMD, SonarQube, and SpotBugs. Due to the great number of modifications, we decided to execute a second manual evaluation of 500 projects to test the introduced changes. The LinterDetectionTool did not create findings for 207 projects; we expect these projects not to use a linter. Considering the thresholds we established in the first manual evaluation, we believe 228 projects to be using at least one linter. The LinterDetectionTool did produce findings for the remaining 65 projects, but we exclude them from the study according to our established thresholds. The results for the individual linters are displayed in Table 3.4.

We randomly manually evaluated 100 of the projects that we consider to be using linters. We were not able to detect a single false positive among these projects. This indicates

[13] `https://github.com/qos-ch/slf4j`
[14] `https://www.microfocus.com/en-us/cyberres/application-security`
[15] `https://fortify.net/`

an acceptable accuracy of the LinterDetectionTool in the detection of the considered linters to us. Further adaptation of the signals did, therefore, not appear to be necessary.

Linter	Threshold	Meet Threshold (Would be a study object)	Fail to Meet Threshold (Would be excluded from the study)	Total
Checkstyle	3	139	24	163
FindBugs	4	97	73	170
SpotBugs	5	39	8	47
PMD	2	38	24	62
SonarQube	3	39	20	59
CodeQL	1	19	0	19
Codacy	1	17	0	17
Coverity	1	7	0	7

Table 3.4: Final list of linters included in the study with number of findings required for inclusion in the study (threshold considered in the third post-analysis exclusion criteria) and outcome of the second manual evaluation.

Discussion and Conclusion

The evaluation shows that an explicit automatic assessment of the integration of a static analysis tool in a GitHub project is impossible. We can, therefore, only develop a tool that tries to detect linters in projects with reliable accuracy. To increase the accuracy of the LinterDetectionTool, we can introduce a minimum necessary quantity of detected signals for each linter (threshold).

It is even harder to state that a project does not use a SAT. Some linters might be utilized in the development process without being visible in the repository. There could, for example, be an oral agreement among developers that they run linters via their CLIs' on their computers before accepting PRs for a project. Additionally, the linters and integration possibilities for linters detected by the LinterDetectionTool are very likely not exhaustive. This is especially true since there is a tremendous amount of possibilities to integrate a SAT into the development life cycle as well as a large variety of linters available.

Hence developers of a project for which the LinterDetectionTool produces zero findings might still use a SAT. Therefore the result of the tool is only our most accurate guess

regarding the integration of a linter in a project, and we can not automatically assess if the developers are using a linter and further resolving its generated warnings.

3.6 Code Quality Analysis Procedure

In this section, we describe the details of our static code analysis implemented using Teamscale.

Analysis Procedure Overview

We analyse a randomly selected set of projects satisfying our pre-download requirements with TS, always using the same analysis profile. After cloning a project, a TS analysis is triggered to analyse the cloned project stored in our local file system. After analysis, we save all considered quality metrics to a database for later evaluation.

Source Code Exclusion

Some code in software projects is less important for the functionality of a system than other [Vei16]. Therefore why we decide to exclude specific code from our analysis. We copy the approach established by Veihelmann in [Vei16]. Therefore we try to automatically exclude all test code, generated files, and third-party libraries using path-based exclusion and the detection of test frameworks. TS allows the configuration of both exclusion mechanisms on project creation. The patterns used to exclude paths as well as test frameworks can be found in the Appendix under *Path-based Exclusions* and *Test Framework Checks*.

Code Metric Settings

The following paragraphs describe the settings of the TS analysis profile we used to assess the code quality metrics listed in Section 2.1. An overview of the analysis profile is displayed in Table 3.5.

Clone Coverage TS calculates the numeric metric clone coverage for our projects. We consider a minimal clone length of 10 statements to be a suitable threshold, aligning with values used in similar studies [Vei16; Jue11]. TS detects clones of type-2; thus, changing names of identifiers will not prevent a clone from being found.

Comment Completeness TS automatically detects relevant interface comments. We configure TS to not expect interface comments for simple getters, setters, and overwritten methods. We do expect Javadoc comments for all public types, methods, and attributes. Javadoc comments are delimited by the `/**...*/` delimiters. TS directly calculates the numeric metric comment completeness for us.

Nesting Depth We have configured TS to create a method-based assessment metric for nesting depth. Our configured thresholds are 4 for yellow and 5 for red. These thresholds are a little more strict than thresholds proposed in some papers that consider five as a yellow threshold [Sch+15; Vei16]. We calculate the percentage assessment for evaluation purposes ourselves. Since the assessment is method based, y_p is the percentage of methods with a maximum nesting depth greater than or equal to 4, and r_p is the percentage of methods exceeding a nesting depth of four.

Method Length The method length assessment is configured almost equivalent to the nesting depth assessment. Our chosen thresholds for method length are 30 statements for yellow and 50 for red. These values lie within the ranges of analysis profiles used for similar evaluations [Sch+15; Vei16]. The calculation of the percentage assessment is accordingly carried out in the same way as for nesting depth. Hence y_p is the percentage of methods with a length greater than or equal to 30, and r_p is the percentage of methods with length greater than or equal to 50.

File Size The file size assessment is also configured almost identical to nesting depth and method length. The major difference is that we calculate the assessment based on SLOC. Our selected thresholds are 300 SLOC for yellow and 500 for red. These values also align with thresholds considered by Veihelmann in [Vei16] and Schüler in [Sch+15]. Accordingly, we calculate y_p as the percentage of SLOC within files with more than or equal to 300 SLOC and r_p with length more than or equal to 500 SLOC.

Metric	Setting	Value
Clone Coverage	Statement-Based	true
	Threshold	10
Comment Completeness	Required interface comments	public & (type | method | attribute) & !(simpleGetter | simpleSetter | annotated(Override) | override)
	Must be Javadoc	true
Nesting Depth	Method-Based	true
	Thresholds	4, 5
Method Length	Method-Based	true
	Thresholds	30, 50
File Size	SLOC-Based	true
	Thresholds	300, 500

Table 3.5: TS analysis profile settings

4 Case Study Results and Interpretation of Findings

This chapter presents the results of the analyses we performed to answer the research questions and hypotheses introduced in the previous chapter. We provide insights on overall linter distributions and compare code metrics between projects with and without linters. Additionally, we will interpret the results of our analysis in the course of this chapter.

Number of Study Objects

We analysed a total of 5,911 projects. Applying post-analysis exclusion criteria 1 & 2 (under 1,000 SLOC & above 75% clone coverage) excludes 278 of our projects, leaving us with 5,633 projects. Applying post-analysis exclusion criteria 3, excluding projects where the LinterDetectionTool produces findings that do not exceed our in Section 3.5 established threshold for any linter, leads to the exclusion of another 616 projects. Thus we have a total of 5,017 study objects. According to the results of the LinterDetectionTool, we have 847 projects that use at least one linter and 4,170 that do not use a linter.

4.1 RQ 1: Distribution of linters in open-source software projects

This research question aims to gain insights into the distribution of linters in OSS systems.

Overall Linter Distribution

As a brief introduction, we show the frequency of each linter recognized by the LinterDetectionTool among our 5,017 study objects (projects after applying post-analysis criteria) to give the reader an idea of how popular the various linters are.

Results

We have 847 projects where we detected at least one linter; the linters individual frequencies are depicted in Figure 4.1. Checkstyle is by far the most frequently used linter being integrated into 63% of the 847 projects, followed by FindBugs, which is incorporated into 40% of the projects. SonarQube was detected in 155 of the 847 projects

(18%) and is the third most commonly used static analysis tool. The LinterDetectionTool detected PMD in 86 of our 4,941 projects; this accounts for 10% of our projects in which we detected a linter. Less frequently used are the tools Codacy, Coverity, and CodeQL.

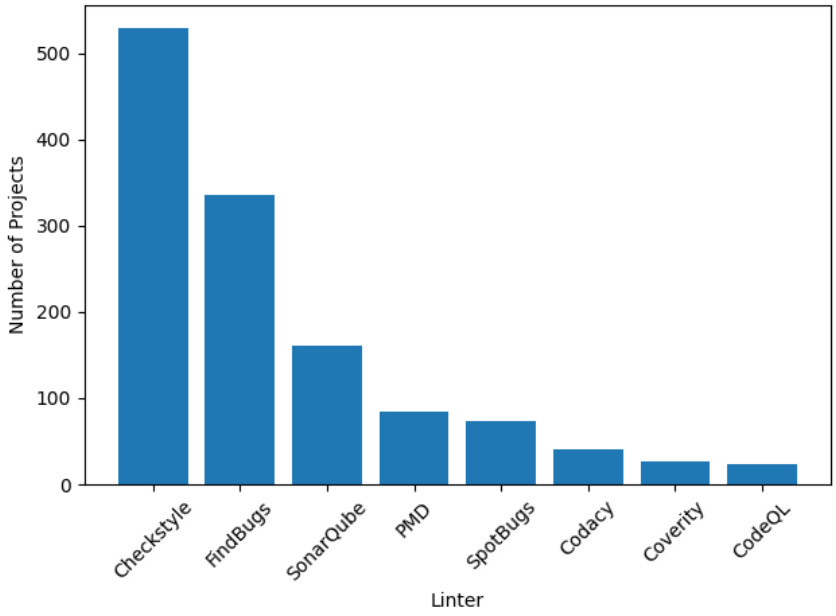


Figure 4.1: Distribution of linters in our study objects

Interpretation

This distribution shows that a lot of teams are interested in making developers adhere to coding standards. Fewer teams seem to be interested in integrating finding bugs. Developers agree that coding-style violations are the most frequently created warnings by static analysis tools [NWA20]. On the other hand, developers wish for more security and bug-related warnings [NWA20]. Therefore a possible reason for this distribution, since there is a discrepancy among the desires of the individual developers and the integrated SATs, is marketing and overall popularity of the linters.

We also observe that a lot fewer projects use SpotBugs than FindBugs. Considerably this less frequent occurrence of SpotBugs happens since we did not take the recent development history of the projects into account, and project development may have stopped before SpotBugs succeeded FindBugs. Therefore the distribution might also not be representative when aiming to evaluate the most recent trends.

It also seems unlikely that tools which occur more frequently in our dataset are superior to less frequently used tools, primarily since Checkstyle, the most commonly used linter, cannot determine complete inheritance hierarchies or detect clones.

H 1: *Linter frequency in relation to project size*

Since maintenance effort increases with increasing project size, a need for high software quality rises. We, therefore, expect to find linters more frequently among larger projects. To verify this hypothesis, we compare the relative number of projects with linter(s) detected by the LinterDetectionTool in the three project size groups small, medium, and large.

Results for *H 1*

Small projects Table 4.1 shows that the LinterDetectionTool classifies around 82% of small projects as not using a linter. Following our evaluation of the LinterDetectionTool (Section 3.5) and the established thresholds, 9% of the projects are considered ambiguous, and 9% use at least one linter.

Medium projects The LinterDetectionTool generates no findings for 66% of the medium-sized projects. This percentage is a remarkable decrease compared to small projects. The thresholds established in Section 3.5 make us mark 13% of the medium-sized projects as ambiguous, while we are confident that a linter is used in 21% of them.

Large projects The LinterDetectionTool creates no findings for 30% of our large projects. This percentage is remarkably smaller than the percentages of projects with zero findings for medium and small projects. We again observe a decrease in the relative number of projects excluded from the study due to their low number of findings. We are unsure about linter usage in 20% and very sure that a linter is being used in 50% of large projects. Including 50% of large projects in our study is a major increase compared to medium-sized projects (21%) and an even greater increase from small projects (9%).

In summary, we do observe a clear increase in linter detections with rising project size. Therefore, we conclude:

The more SLOC a project has, the higher the likelihood that a linter is being used.

Size in KSLOC	Total	No Linter	Ambiguous	With Linter(s)
1 to 10 (small)	3,529	2,882 (82%)	323 (9%)	324 (9%)
10 to 100 (medium)	1,845	1,236 (66%)	242 (13%)	392 (21%)
over 100 (large)	259	77 (30%)	51 (20%)	131 (50%)

Table 4.1: Linter detection distribution in correlation to project size after application of post-analysis exclusion criteria 1 & 2

Interpretation of the Results of *H 1*

The results confirm our assumption. Therefore we presume that developers integrate SATs when the code base reaches a size that leads to a complexity that is difficult to grasp by a single developer. The observations made by Bhatia that increasing SLOC leads to more complexity and cost support our assumption [BM14].

Another observation we make is the decreasing elimination of projects due to the thresholds established in Section 3.5. We have two explanations for this occurrence. The first is that we could lower the thresholds for small projects since they are likely to have fewer mentions of the linters in code comments simply due to their lower SLOC without decreasing the tool's accuracy. On the other hand, we chose the thresholds close to the number of findings caused by integration mechanisms. Therefore, we see the second cause being developers testing linters on a trial basis but not integrating them into the projects. We assume that a combination of both assumptions is the case.

H 2: *Linter frequency in correlation to team size*

This hypothesis assumes that core developers see a need for static analysis tools to reduce review efforts and increase code quality with an increase in overall contributors. To rule out possible effects of project size on team size, we analyze the correlation between the number of contributors and the use of linters in the size groups introduced in *H 1*. We only consider our 5,017 study objects to evaluate this hypothesis.

Results for *H 2*

Small projects The results of *H 1* show that among our study objects, we only detect linters in 9% of the small projects. Separating small projects by team size (Figure 4.2) shows that 1,480 projects are personal and that we only consider linter usage in 6.2% of them. We also have a large number of projects (1,213) with small team sizes from 2 to 5; among these projects, we consider 9.2% to be using a linter. We then see an almost

steady increase in linter detection likelihood within our selected groups from personal projects to 75% in small projects with 51 to 60 contributors. Then there is a drop to 33.3% for small projects with 61 to 70 developers. This drop is followed by an increase to 60% for small projects with 71 to 80 collaborators. In the final group of small projects with above 80 collaborators, we detect linters in 57.1% of them. Since small projects with over 40 collaborators are rare, our collected data might not be sufficient to contain representative values for the individual groups. But we can consistently observe an over 30% likelihood for linter integration for small projects with over 20 collaborators.

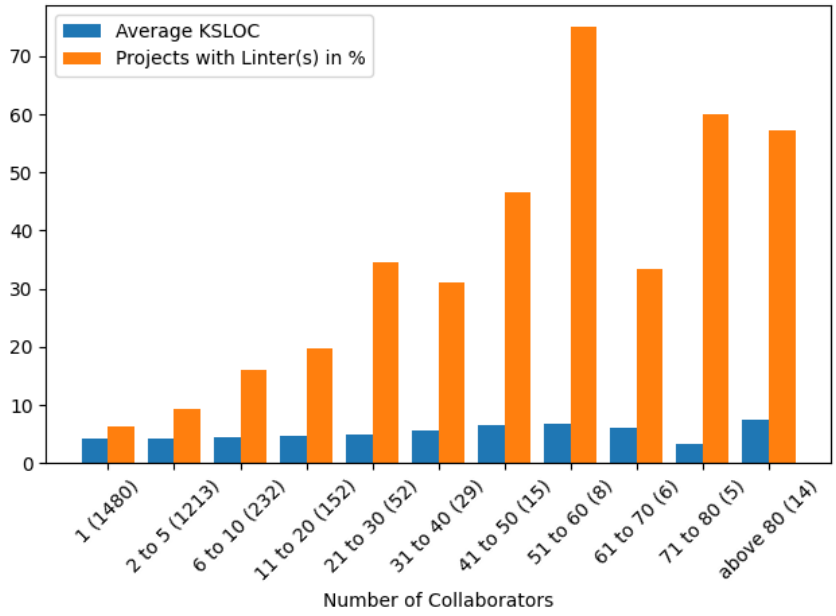


Figure 4.2: Linter detection in relation to team size for small projects

Medium projects In Figure 4.3, we can observe similar behavior for medium as for small projects. Projects in this group are still likely to be personal projects (450) or developed by small teams of 2 to 5 developers (498). We detect linters in 9.6% of medium-sized personal projects and in 13.7% of medium-sized projects with 2 to 5 developers. We then observe a consistent increase in linter detection rate from 9.6% for medium personal projects to 55.6% for medium projects with 51 to 60 contributors. We then observe a drop to 47.2% within the medium project group with 61 to 80 contributors. The percentage points increase again until our last group of medium projects with over 80 contributors, yielding the highest linter detection likelihood for medium projects at 74.7%.

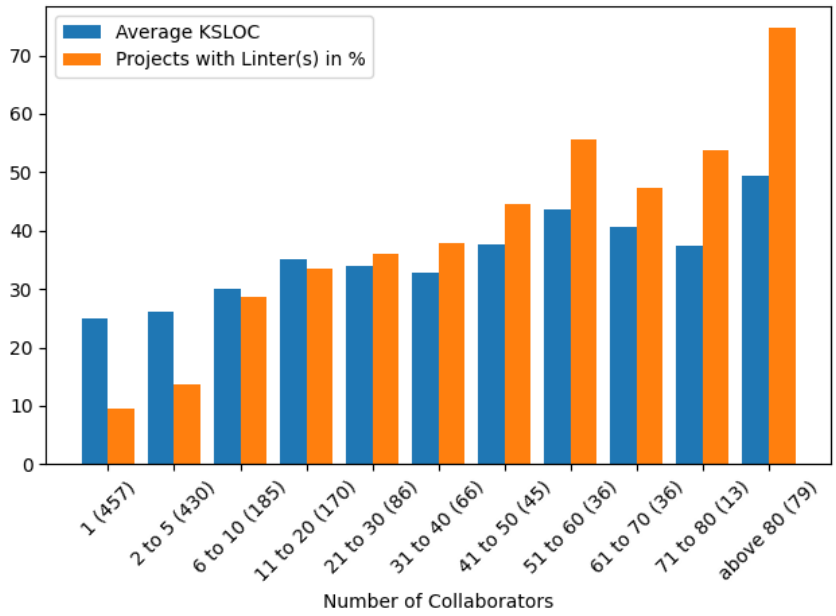


Figure 4.3: Linter detection in relation to team size for medium projects

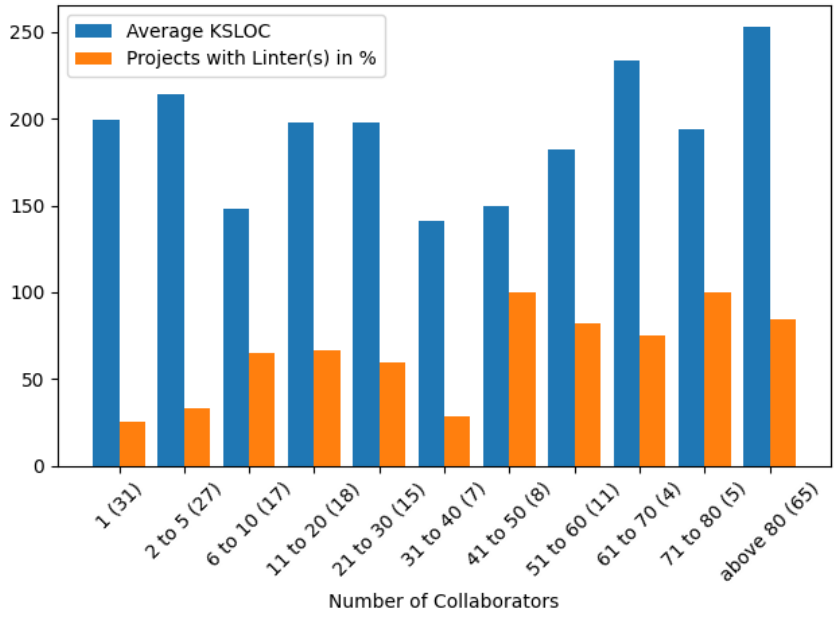


Figure 4.4: Linter detection in relation to team size for large projects

Large projects We have a smaller absolute amount of large projects (208). Figure 4.4 clearly shows that the likelihood for the LinterDetectionTool to detect a linter for large projects under 40 collaborators is lower than for large projects above 40 collaborators. We consider 25.8% of the 31 large personal projects to be using a linter; this is considerably higher than for small and medium personal projects. Taking into account the small number of projects within the project groups separated by the number of collaborators, such as large projects with 41 to 50 or 71 to 80 collaborators with only 8 and 5 projects and a 100% linter detection rate, we do not consider the results to be very representative of the individual groups.

All projects Figure 4.5 shows that if we consider all projects and group them by team size, we can also see a close to consistent increase in the percentage points at which we detect linters with an increasing number of collaborators. It also shows a correlation between team and project size as we observe increasing average KSLOC numbers with increasing team size. This increase is also observable in Figure 4.3 for medium projects but not for small or large projects (Figures 4.2 & 4.4).

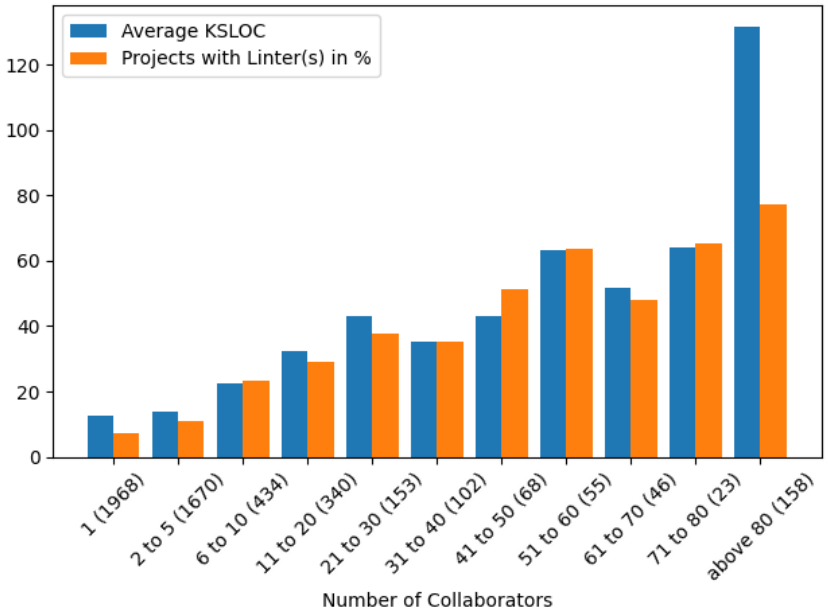


Figure 4.5: Linter detection in relation to team size

For the individual groups as well as for the overall distribution in Figure 4.5, we can observe that with a rising number of collaborators, the likelihood of a linter being in-

tegrated into a project increases. We also observe that despite a general increase in linter detection rate from left to right, there are fluctuations within our selected team size differentiations. We conclude:

> The likelihood of a linter being integrated into a project increases as the number of collaborators increases.

Interpretation of the Results of *H 2*

We can observe a correlation between the number of developers participating in a project and the likelihood that a linter is integrated, regardless of the project's size. As team size increases, so does the number of peripheral developers increases. Therefore we assume that core developers integrate linters into the PR review process to improve the quality of the contributions made by peripheral developers. Thus with increasing team size, there is a higher likelihood for a linter integration within projects. The results also indicate that team size has a more significant impact on the integration of a linter than the size of the repository. Still, the size is not a neglectable factor when it comes to the likelihood of linter integration.

H 3: *Linters rarely come alone*

Since different linters have different benefits, we assume that projects with one linter are likely to use multiple linters. Evaluating this assumption leads to the following result:

Results for *H 3*

The distribution for the number of linters in linter projects can be found in Figure 4.6. From our total of 847 projects that use a linter, a total of 570 (67.3%) use a single linter. Therefore we have a total of 277 (32.7%) projects that use more than one linter. Combinations of two to three linters are quite common (29.4%), while combinations of more than three linters are very rare (3.3%).

The most common combination among projects using linters is Checkstyle and FindBugs occurring in 20.2% of projects using a linters (Table 4.2). Other common combinations are Checkstyle and PMD (8.1%) and Checkstyle and SonarQube (6.7%). PMD is very rarely used by itself, as we only detected 7 projects that solely use PMD. Additionally, we observe that a combination of FindBugs and SpotBugs is present in 6.1% of our projects with linter integration. The combination of Checkstyle, FindBugs, and PMD occurs in 41 (4.9%) of our projects using linters.

32.7% of projects using linters use multiple linters; however, this is not a majority. Therefore we discard our assumption and conclude:

> The majority of projects with linter integration integrate exactly one linter. One-third of the projects that integrate a linter integrate more than one linter.

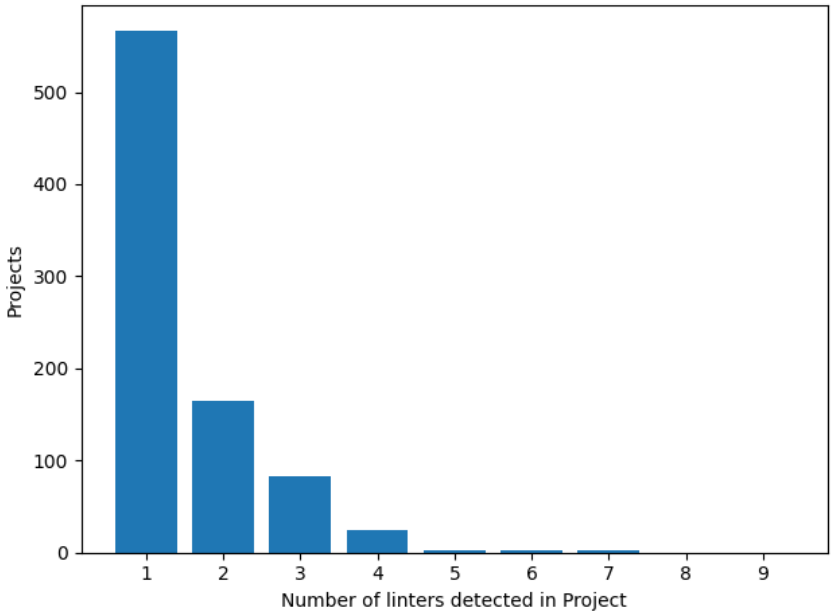


Figure 4.6: Number of linters used in projects

Interpretation of the Results of *H 3*

Despite rejecting our original assumption for *H 3*, our results show that the integration of multiple linters is not unlikely. The results show that close to one-third of projects with linter integration have multiple linters integrated. We conclude that one third of OSS development projects see the divergent benefits of the different linters and want to increase their likelihood of detecting "code smell" early by combining them. The high combination of projects using Checkstyle and FindBugs/SpotBugs supports this. Other combinations, like Checkstyle and PMD, show that there also exists interest in using

Linters	Total	Relative
Checkstyle, FindBugs	169	20.2%
Checkstyle, PMD	68	8.1%
Checkstyle, SonarQube	56	6.7%
Checkstyle, SpotBugs	53	6.3%
FindBugs, SpotBugs	51	6.1%
PMD, FindBugs	47	5.6%
Checkstyle, FindBugs, PMD	41	4.9%
Checkstyle, FindBugs, SpotBugs	39	4.7%
SonarQube, FindBugs	32	3.8%

Table 4.2: The most common combinations of linters. The relative amount stands in relation to the total number of projects using a linter (837)

different linters with overlapping checks. Overall the majority of projects using a linter uses just one linter.

Also, we observe multiple projects in which we detect both FindBugs and SpotBugs. These projects have most likely been updated from FindBugs to SpotBugs.

RQ 1: Interpretation of Findings

The evaluation of the results from *H 1* & *2* shows that there are correlations between project size, team size, and linter integration. The results from *H 2*, especially in Figure 4.2 for small projects, indicate that the number of collaborators is potentially a more decisive factor for the likelihood for a linter in a project. Since larger teams are more likely to integrate a linter, we assume that (core) developers utilize linters for quality assurance in new contributions (from less experienced or peripheral developers).

The results from *H 3* show that one-third of the projects with linter integration utilizes multiple linters. We assume that they aim to improve their software's quality by using the divergent benefits of the different linters.

4.2 RQ 2: Impact of linters on software quality

This research question aims at providing insights regarding the distributions of our examined source code metrics while taking the detection of linter(s) into account.

In the following paragraphs, we present and describe the results created by our static code analysis with Teamscale and the LinterDetectionTool of the downloaded projects.

Interpretation of Figures We use boxplots to compare the metrics in the different groups. For numeric metrics, we directly utilize the calculated values of the projects as plot inputs. The respective percentage assessment value is used as plot inputs for assessment metrics. We will compare yellow and red percentage assessments individually. Lower boxes correspond to better metric values.

Interpretation of Tables Each boxplot comparison will have corresponding table entries with the means, medians, and p-value produced by the respective Wilcoxon test. If the p-value produced by the Wilcoxon test is higher than 1%, the distributions are not considered to be significantly different.

H 4: *Code Clones*

This hypothesis evaluates whether we can observe significant differences in the relative amounts of code duplicates among projects with and without linter integration. We have a total of 161 projects using SonarQube and 84 projects using PMD. We only have 7 projects using PMD and SonarQube, resulting in 238 study objects to evaluate *H 4*.

Results for *H 4*

As visible in the boxplots in Figure 4.7, projects with linters from all four comparisons seem to have fewer code clones than projects without linters. The Wilcoxon tests (p-values in Table 4.3) show that the clone coverage distributions are significantly different in three of our four comparisons. Therefore small and medium projects (Figure 4.7: (a) and (b)) show a significantly superior clone coverage for projects using linters. According to the Wilcoxon test for comparing clone coverage in large projects with $(p > 0.01)$, we do not consider the clone coverage in large projects as significantly different. Figure 4.7 (c), as well as the difference in means of $|9.0\% - 14.1\%|$ = 5.1% percentage points, still indicate superiority in favor of large projects using linters.

We also observe that the worst medium project with a linter has 39% clone coverage while the worst medium project that does not integrate a linter has almost 75% clone coverage. We observe similar behavior for large projects where the worst large project

with a linter has 46% clone coverage. In contrast, the worst large project without linter is also close to 75% clone coverage.

For projects without linters, the differences in medians are $|8.1\% - 11.8\%|$ = 3.7% percentage points from small to medium and 2.3% percentage points from medium to large, leaving us with an absolute difference of 6% percentage points for projects without linters. For projects using linters the differences in medians are 2.5% percentage points from small to medium and 2.7% percentage points from medium to large. This leads to a total difference of 5.2% percentage points which is slightly better than for projects without linters.

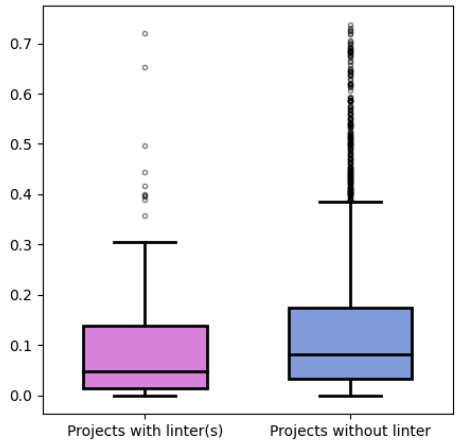
Finally, comparing all projects (Figure 4.7 (d)) with a mean difference of 4.2% percentage points highlights the supremacy of projects using linters.

Therefore we conclude:

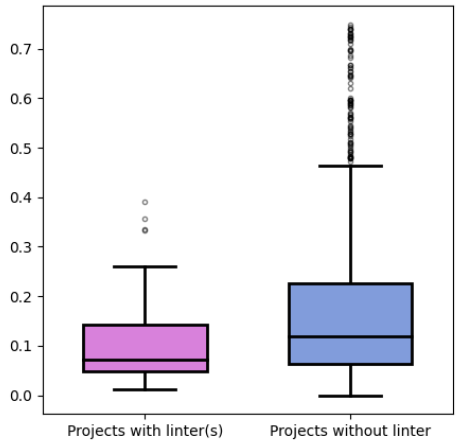
> Small and medium projects using linters that support code clone detection have *significantly* fewer code clones than projects not using a linter. For large projects, the tests show similar advantages, but they are not statistically significant.

Set Name	**Set Size**	**Avg. KSLOC**	**Mean**	**Median**	p**-value**
Projects with linter(s) (small)	108	5.0	10.4%	4.8%	$7 \cdot 10^{-4}$
Projects without linter (small)	2828	4	13.0%	8.1%	
Projects with linter(s) (medium)	102	38.0	10.2%	7.3%	$7 \cdot 10^{-6}$
Projects without linter (medium)	1211	27.0	17.4%	11.8%	
Projects with linter(s) (large)	28	194	13.3%	9.0%	0.04
Projects without linter (large)	77	213	20.3%	14.1%	
Projects with linter(s) (all)	238	41	10.7%	6.9%	$1 \cdot 10^{-4}$
Projects without linter (all)	4170	15	14.5%	9.3%	

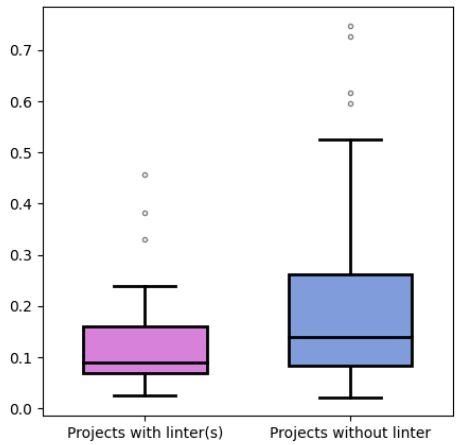
Table 4.3: Clone coverage (Numbers accompanying Figure 4.7)



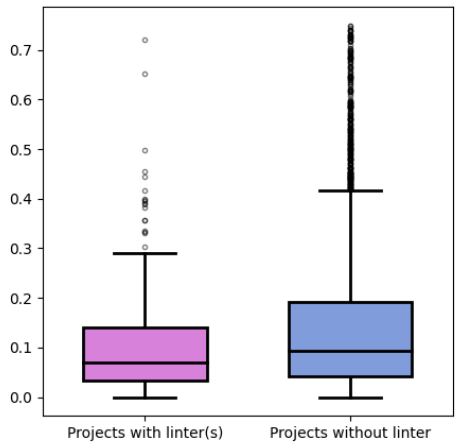
(a) small



(b) medium



(c) large



(d) all

Figure 4.7: Clone coverage comparisons

Interpretation of the Results of *H 4*

Like Veihelmann in [Vei16], we can observe higher clone coverage values with increasing project SLOC. Since the 5.2% percentage point increase in medians from small to large projects with linters is only slightly better than the 6% increase for projects without linters, we conclude that linters can help reduce clone coverage but can not prevent decay with increasing project SLOC.

Heitlager *et al.* consider a median of 5% clone coverage as a reasonable value for well-designed systems [HKV07]. We observe a slightly higher median of 6.8% among the systems that use linters. The median for systems not using a linter of 9.3% is higher. It, therefore, increases the probability of quality issues due to code clones in projects that do not use a linter compared to projects using linters. According to Heitlager *et al.*, source code erosion is out of control in systems with clone coverage above 20% [HKV07]. Considering that Heitlager *et al.* considered code clones with 6 SLOC, whereas we detect duplicates starting at 10 SLOC, we can speculate that some large projects not using a linter exceed this threshold.

From our findings, we conclude that linters are an efficient tool to detect and reduce code clones. Despite the improvements linter usage achieves, we can still observe a relatively high number of code clones above 6% in medium and large projects. An explanation could be that refactoring may not always improve software regarding clones since some clones might be short-lived and diverge from one another. Due to programming language limitations, long-lived clones that have changed consistently with other elements in the same group might be difficult to refactor [Kim+05]. Therefore some clones might be detected but purposely not resolved by the developers.

H 5: *Comment Completeness*

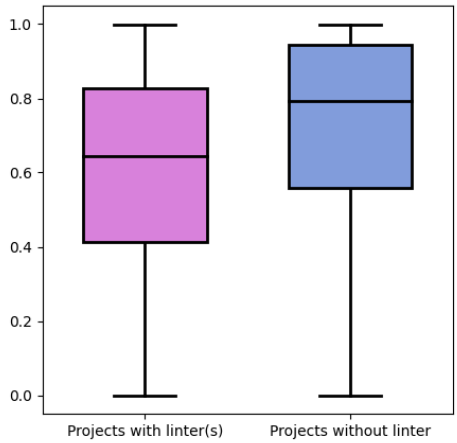
This hypothesis evaluates whether we can observe significant differences between the interface documentation among projects using and not using linters. For hypotheses *5* to *8*, we consider projects with linters Checkstyle, SonarQube, PMD, Codacy, and Coverity, resulting in 683 projects with these linters from which 265 are small, 314 are medium, and 104 are large projects using linters.

Results for *H 5*

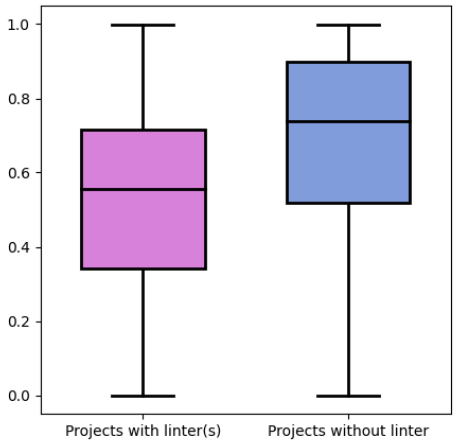
Figure 4.8 (a), (b), and (d) show that the comparisons for small, medium, and all projects show significantly better ($p < 0.01$) comment completeness when a linter is integrated. Similar to clone coverage, the determined p-value for large projects (see Table 4.4) is greater than 0.01; therefore, we do not consider the results statistically significant. Figure 4.8 (c) still shows a clear tendency for better comment completeness in favor of projects using linters.

Since in three of our four comparisons, projects with linters show significantly better comment completeness, we conclude:

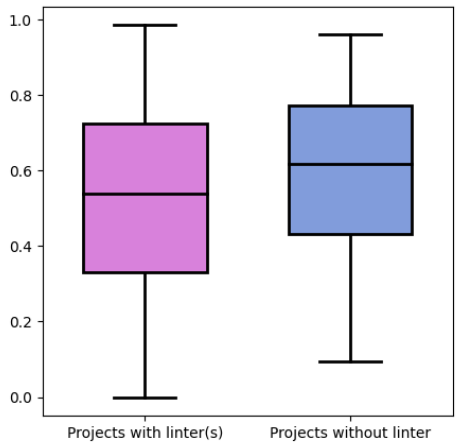
> Small and medium projects using linters have statistically significant advantages over projects not using a linter for comment completeness. For large projects, the tests show similar advantages, but they are not statistically significant.



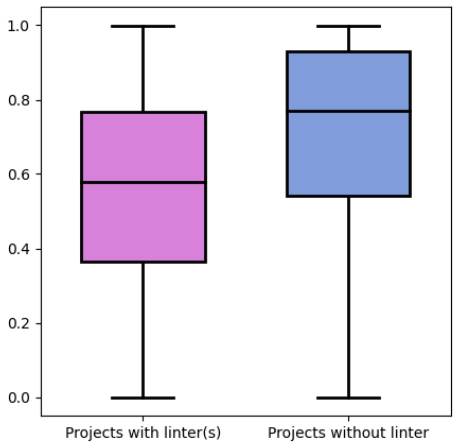
(a) small



(b) medium



(c) large



(d) all

Figure 4.8: Missing interface comments comparisons

Set Name	Set Size	Avg. KSLOC	Mean	Median	p-value
Projects with linter(s) (small)	265	5	60.2%	64.3%	$3 \cdot 10^{-11}$
Projects without linter (small)	2828	4	71.9%	79.3%	
Projects with linter(s) (medium)	314	37	52.7%	55.6%	$1 \cdot 10^{-21}$
Projects without linter (medium)	1200	27	68.3%	73.9%	
Projects with linter(s) (large)	104	203	52.2%	54.1%	0.04
Projects without linter (large)	77	213	59.7%	61.8%	
Projects with linter(s) (all)	683	50	55.5%	57.9%	$3 \cdot 10^{-43}$
Projects without linter (all)	4105	15	70.6%	77.1%	

Table 4.4: Missing Interface Comments (Numbers accompanying Figure 4.8)

Interpretation of the Results of *H 5*

Like Steinbeck and Koschke in [SK21], we observe that the comment completeness is relatively low for a majority of all analysed projects. Like Veihelmann in [Vei16], we observe that comment completeness improves with project size. Despite the better comment completeness in projects using linters, a median of 42.1% ($= 1 - 57.9\%$) is still not very high.

Studies have shown that poor documentation significantly lowers the maintainability of software [Lie83], but commenting code is often neglected due to release deadlines and other time pressure during development [SHJ13]. Depending on project PR acceptance criteria, linters can remind or enforce developers to write interface comments.

Looking at the boxplots for all comparisons in Figure 4.8, we can see that only a view projects have 100% comment completeness. Additionally, we observe that all large projects with 100% comment completeness use a linter. Since missing interface documentation is easy to detect during the code review process, teams could try to enforce interface documentation without the help of a linter.

Therefore, we conclude that linters can help remind developers to write crucial interface comments, but only very few projects seem to enforce documentation by using a linter.

H 6: *Nesting Depth Violations*

This hypothesis evaluates whether we can observe significant differences between nesting depth violations among projects using and not using linters.

Results for *H 6*

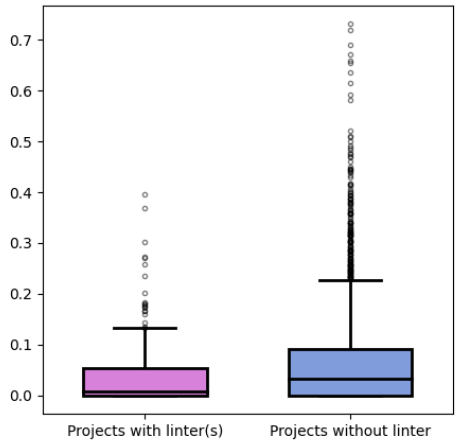
For both thresholds, yellow and red, supported by Wilcoxon tests (Table 4.5), the comparisons for small and medium projects show fewer nesting depth violations within projects using linters. Based on the p-values for large projects ($p > 0.01$) for both yellow and red, we do not consider the distributions significantly different. The boxplots for large projects in Figures 4.9 (c) and 4.10 (c) both show a tendency for lower nesting depth violations within projects using linters.

Like Veihelmann in [Vei16], we observe that larger projects tend to show slightly more inferior quality in terms of nesting depth. Due to the overall size distribution and the detected correlation between size and likelihood for linter integration, the boxplots, including all projects (Figures 4.9 (d) and 4.10 (d)), can be very misleading.

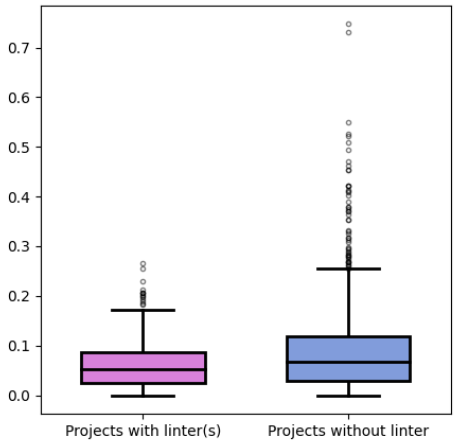
The Wilcoxon tests (Table 4.5) for our comparisons, including all projects, do not consider the distributions for red and yellow statistically significant. As stated in our evaluation approaches for *H 4* to *H 8* in Section 3.3, comparisons including all projects are not considered when it comes to drawing a conclusion from our findings.

We conclude:

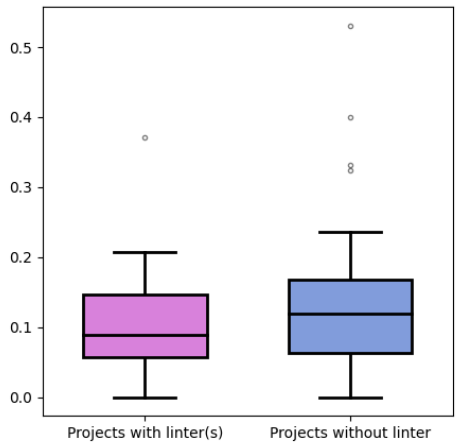
> Small and medium projects using linters have statistically significant but small absolute advantages over projects not using a linter for nesting depth. For large projects, the tests show similar advantages, but they are not statistically significant.



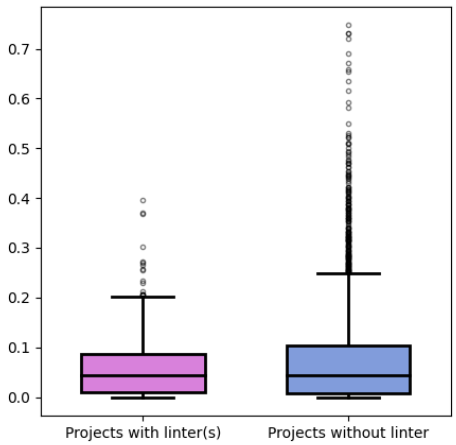
(a) small



(b) medium

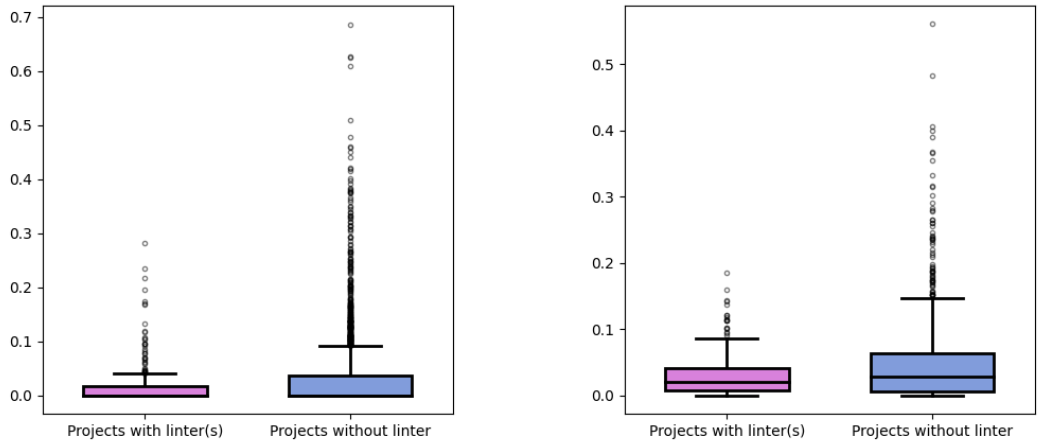


(c) large



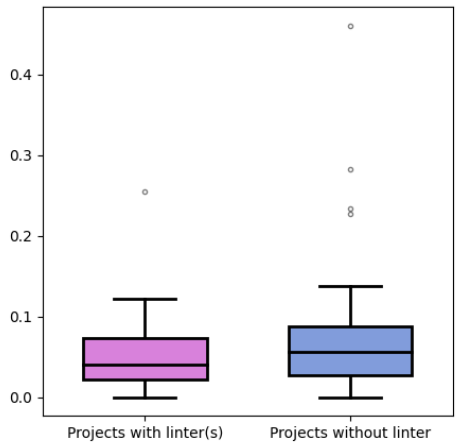
(d) all

Figure 4.9: Nesting depth violation comparisons with yellow percentage assessment

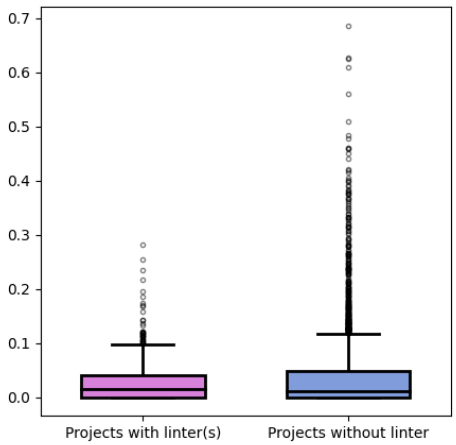


(a) small

(b) medium



(c) large



(d) all

Figure 4.10: Nesting depth violation comparisons with red percentage assessment

Set Name	Set Size	Avg. KSLOC	Mean	Median	p-value
Projects with linter(s) (small)	265	5	3.9%	0.8%	$8 * 10^{-9}$
Projects without linter (small)	2828	4	6.6%	3.2%	
Projects with linter(s) (medium)	314	37	6.5%	5.2%	$3 * 10^{-4}$
Projects without linter (medium)	1200	27	8.8%	6.6%	
Projects with linter(s) (large)	104	203	10.0%	8.9%	0.1
Projects without linter (large)	77	213	12.3%	11.9%	
Projects with linter(s) (all)	683	50	6.0%	4.4%	0.3
Projects without linter (all)	4105	15	7.3%	4.4%	
Projects with linter(s) (small)	265	5	1.8%	0.0%	$7 \cdot 10^{-6}$
Projects without linter (small)	2828	4	3.3%	0.0%	
Projects with linter(s) (medium)	314	37	3.0%	2.1%	0.002
Projects without linter (medium)	1200	27	4.6%	2.8%	
Projects with linter(s) (large)	104	203	5.0%	4.0%	0.1
Projects without linter (large)	77	213	6.8%	5.6%	
Projects with linter(s) (all)	683	50	2.8%	1.5%	0.2
Projects without linter (all)	4105	15	3.7%	1.0%	

Table 4.5: Nesting depth violations (Numbers accompanying Figures 4.9 and 4.10)

Interpretation of the Results of *H 6*

Like Veihelmann in [Vei16], we observe that the distributions for nesting depth are notably better than the other code quality metrics. This could be due to increased attention to avoid very nested code regions or point out that nesting depth simply is created more rarely during the "natural process" of programming [Vei16]. We observe that large projects seem to have a higher percentage of methods that violate our thresholds (similar to code clones).

This increase could argue slightly against the assumption that a high nesting depth rarely arises during the "natural process" of programming since the nesting depth is calculated separately for each method. A possible explanation for the increase in very nested code regions could be that peripheral, less experienced developers are more likely to produce methods with higher nesting depth. On the other hand, the more reasonable assumption is that in the case of large, especially growing and frequently expanded projects, several requirements at different times are responsible for the increase in

nesting depth violations.

Similar to code clones, linters decrease the amount of nesting depth violations, but some still remain. Therefore, we expect that resolving deep nesting is left optional and not mandatory for PR acceptance in most projects, similar to missing interface documentation.

H 7: *Method Length Violations*

This hypothesis evaluates whether we can observe significant differences between method length violations among projects using and not using linters.

Results for *H 7*

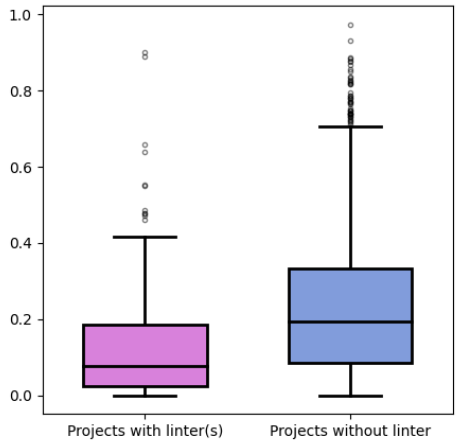
Again our results regarding correlations to size and distribution align with the observations made by Veihelmann in [Vei16]. Similar to clone coverage and nesting depth, the relative number of method length violations (Figures 4.11 and 4.12) increases with increasing project size. We also observe that the share of repositories with (very) good method lengths is, relatively speaking, smaller when compared to the metrics clone coverage and nesting depth. For instance, 50% of the repositories not using a linter come with at least 21.2% of method length violations exceeding our threshold of 30 statements. The median for repositories using linters of 17.2% is 19% lower.

These values decrease when considering our red threshold of 50 statements (Table 4.6).

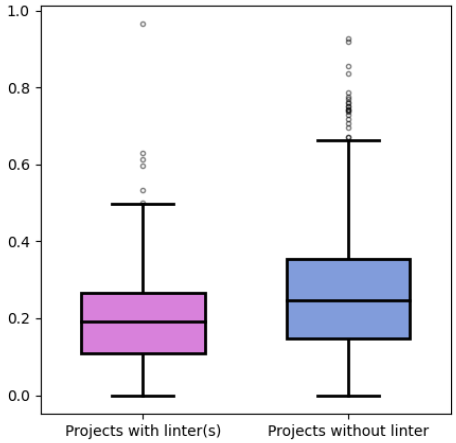
We can confirm that all method length violation comparisons (Figures 4.11 and 4.12), supported by Wilcoxon tests (Table 4.12), are significantly different in favor of the projects using linters. Especially in small projects, using a linter reduces the median for method length violations exceeding 30 statements by 11.7% percentage points and considering our threshold of 50 statements by 7.5% percentage points with a median of 0% for small projects using linters In large projects, the differences in medians are slightly smaller, with an improvement of 6.1% percentage points for the yellow and a 5% percentage point improvement with the red threshold.

Therefore we conclude:

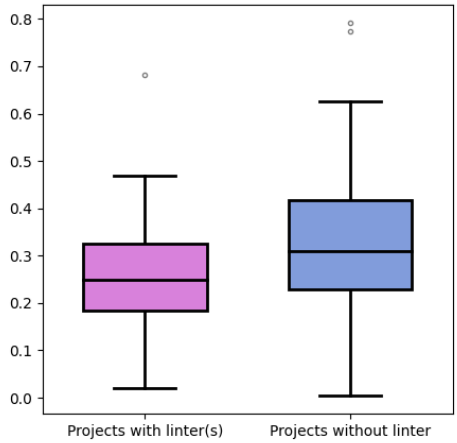
> Small and medium projects using linters that can create warnings for method length violations have *significantly* fewer method length violations than projects not using a linter. For large projects, the tests show similar advantages, but they are not statistically significant.



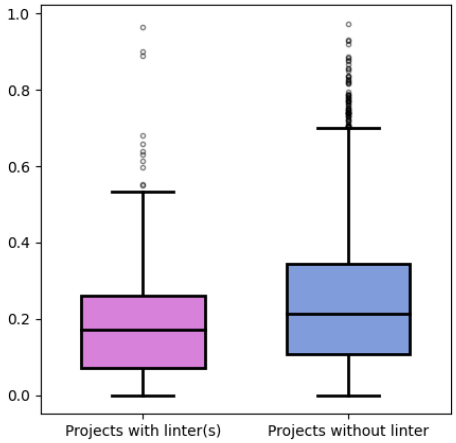
(a) small



(b) medium

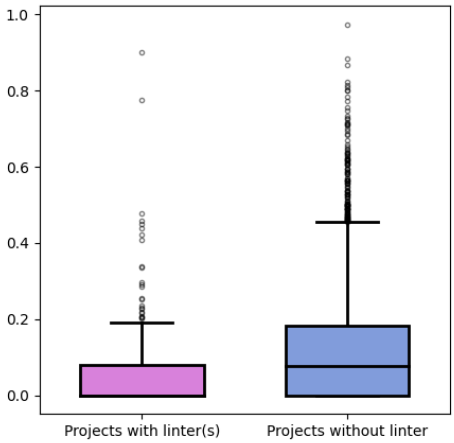


(c) large

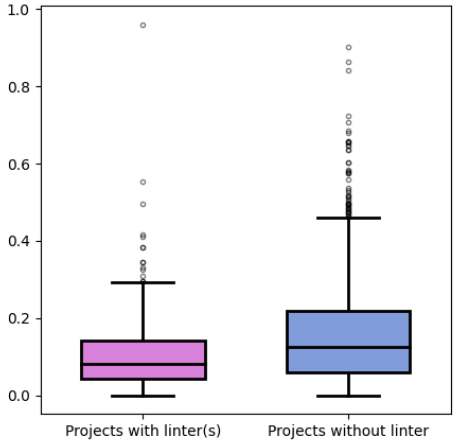


(d) all

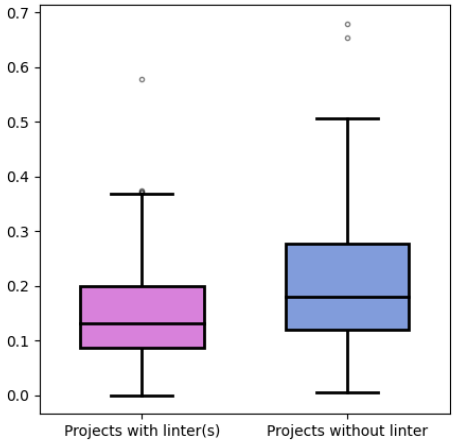
Figure 4.11: Method length violations with yellow percentage assessment



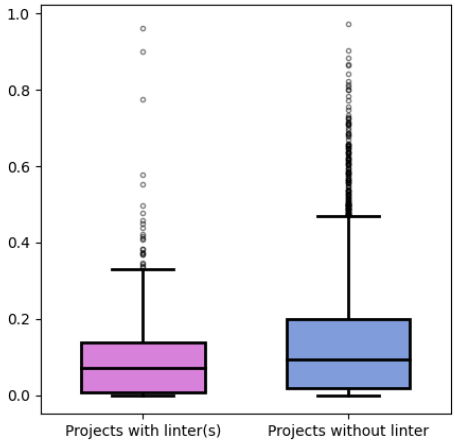
(a) small



(b) medium



(c) large



(d) all

Figure 4.12: Method length violations with red percentage assessment

Set Name	Set Size	Avg. KSLOC	Mean	Median	p-value
Projects with linter(s) (small)	265	5	12.8%	7.7%	$3 * 10^{-22}$
Projects without linter (small)	2828	4	22.8%	19.4%	
Projects with linter(s) (medium)	314	37	20.4%	19.1%	$2 * 10^{-10}$
Projects without linter (medium)	1200	27	26.7%	24.8%	
Projects with linter(s) (large)	104	203	25.9%	24.8%	0.002
Projects without linter (large)	77	213	33.0%	30.9%	
Projects with linter(s) (all)	683	50	18.3%	17.2%	$6 \cdot 10^{-15}$
Projects without linter (all)	4105	15	24.1%	21.2%	
Projects with linter(s) (small)	265	5	6.0%	0.0%	$5 \cdot 10^{-17}$
Projects without linter (small)	2828	4	12.4%	7.5%	
Projects with linter(s) (medium)	314	37	10.7%	8.2%	$7 \cdot 10^{-6}$
Projects without linter (medium)	1200	27	15.7%	12.4%	
Projects with linter(s) (large)	104	203	14.9%	13.0%	$9 \cdot 10^{-4}$
Projects without linter (large)	77	213	21.0%	18.0%	
Projects with linter(s) (all)	683	50	9.5%	7.1%	$2 \cdot 10^{-8}$
Projects without linter (all)	4105	15	13.5%	9.4%	

Table 4.6: Method length violations (Numbers accompanying Figures 4.11 and 4.12)

Interpretation of the Results of *H 7*

Similar to clone coverage, we can observe an impact of linter usage to reduce method length violations. Arguably method length violations are easier to resolve than nesting depth violations where the impact of static analysers was not as big. Despite the great improvement linters seem to cause, the medians for projects using linters are still relatively high for all projects considering our yellow threshold of 30 statements. We can see a great improvement for all projects when considering our threshold of 50 statements. This could indicate that many linters aren't configured to create warnings for method length violations starting at 30 statements but for higher values. Our median value of 0% supports this for small projects using linters, whereas small projects without a linter have a median of 7.5% when considering our red threshold. Similar to our previous metrics, we expect that most linter configurations do not enforce but only suggest resolvement to the developers.

H 8: *File Size Violations*

The final metric we evaluate is violations of file size

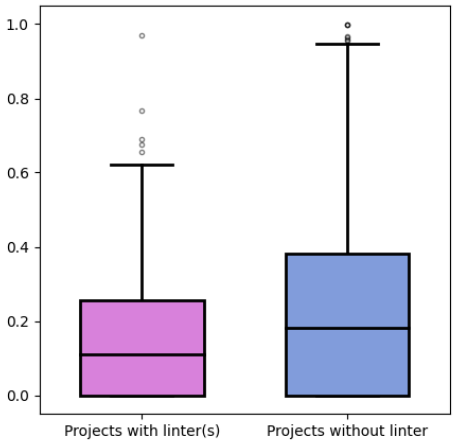
Results for *H 8*

Veihelmann measured modest correlations between violations of file sizes, method lengths, and nesting depth [Vei16]. This indicates that the respective metrics are likely to influence one another, and therefore we expect to see similar results as before. We can also observe that file size violations are even more frequent in medium and large projects for our 300 and 500 SLOC thresholds than method length violations.

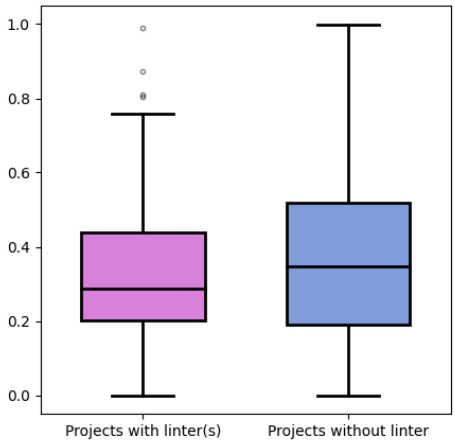
In Figures 4.13 and 4.14, we can instantly see a strong correlation between project size and file size. Therefore our comparisons of all projects (Figure 4.13 (d) and 4.14 (d) are misleading due to the distribution of project sizes on GitHub. According to Wilcoxon tests (Table 4.7), our comparisons for small and medium projects show statistically significant slightly fewer file size violations when using linters for both thresholds. For large projects, both our p-values exceed 0.01; therefore, we do not consider projects with linters significantly superior despite the obvious tendencies showing superiority for projects using linters visible in Figures 4.13 (c) and 4.14 (c).

Four of our six relevant comparisons, supported by Wilcoxon tests, show statistically significant superior file size assessments for projects using linters. We conclude:

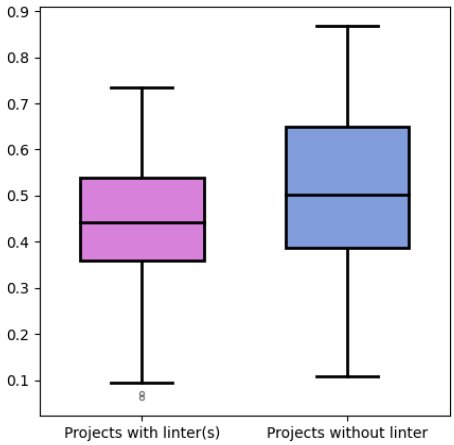
> Small and medium projects using linters that can create warnings for method length violations have *significantly* fewer file size violations than projects without linter. For large projects, the tests show similar advantages, but they are not statistically significant.



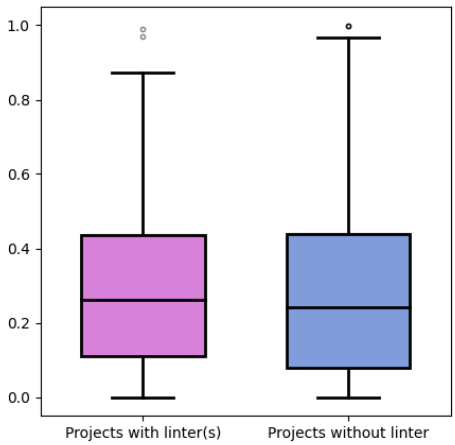
(a) small



(b) medium

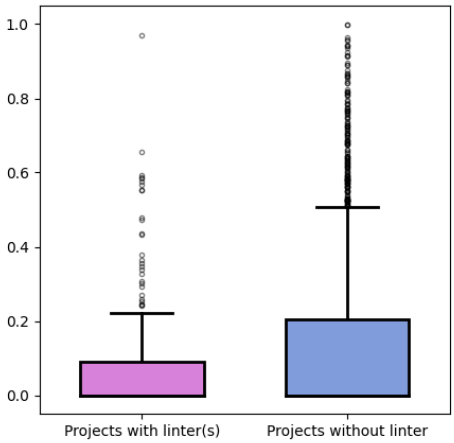


(c) large

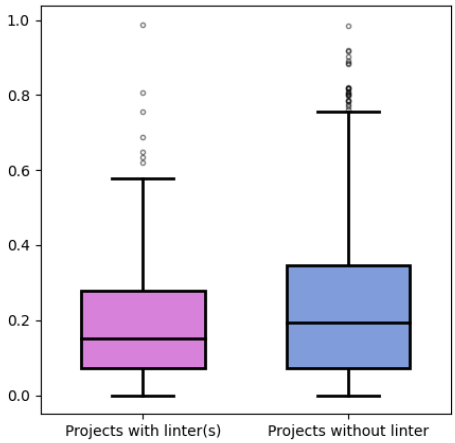


(d) all

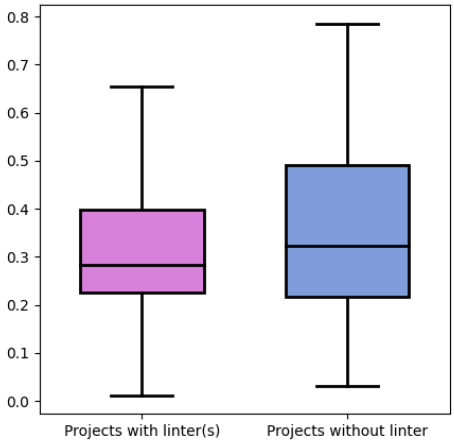
Figure 4.13: File size violation comparisons with yellow percentage assessment



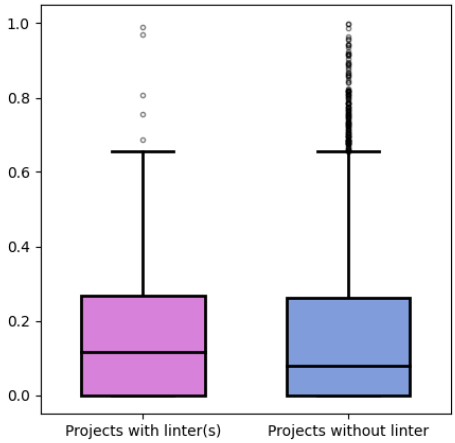
(a) small



(b) medium



(c) large



(d) all

Figure 4.14: File size violation comparisons with red percentage assessment

Set Name	Set Size	Avg. KSLOC	Mean	Median	p-value
Projects with linter(s) (small)	265	5	16.5%	11.0%	$1 * 10^{-6}$
Projects without linter (small)	2828	4	23.6%	18.1%	
Projects with linter(s) (medium)	314	37	32.0%	28.7%	0.001
Projects without linter (medium)	1200	27	36.8%	34.6%	
Projects with linter(s) (large)	104	203	44.4%	44.3%	0.04
Projects without linter (large)	77	213	50.2%	50.1%	
Projects with linter(s) (all)	683	50	27.9%	26.0%	0.2
Projects without linter (all)	4105	15	27.9%	24.2%	
Projects with linter(s) (small)	265	5	7.0%	0.0%	$1 \cdot 10^{-5}$
Projects without linter (small)	2828	4	12.4%	0.0%	
Projects with linter(s) (medium)	314	37	18.8%	15.0%	0.003
Projects without linter (medium)	1200	27	23.7%	19.4%	
Projects with linter(s) (large)	104	203	30.3%	28.3%	0.09
Projects without linter (large)	77	213	35.8%	32.2%	
Projects with linter(s) (all)	683	50	16.0%	11.4%	0.004
Projects without linter (all)	4105	15	16.1%	7.9%	

Table 4.7: File size violations (Numbers accompanying Figures 4.13 and 4.14)

Interpretation of the Results of *H 8*

Especially in large systems, file size violations seem rather common. In large projects, without linters, we find that over 50% of all SLOC belong to files that are more than 300 SLOC long, and over 35% belong to files with more than 500 SLOC. Despite large projects with linters having slightly better medians, 44.3% for our yellow and 28.3% for our red threshold, compared to projects without linters having 50.1% for our yellow and 32.2% for our red threshold, we observe that projects with linters do generally not aim to have very little file size violations. Like for previous metrics, we believe that most linter configurations do not enforce but only suggest resolvement of file size violations to the developers.

RQ 2: Result Summary

Since we observed that projects using linters have significantly better metric distributions for all small and medium comparisons, we can confidently conclude:

> Small and medium projects using linters show *significantly* better code quality than small and medium projects without a linter.

For the majority of comparisons with large projects, our Wilcoxon tests produced p-values above 1%. Still, the boxplots were always in favor of our large projects using linters. Therefore we conclude:

> We observe superior quality metrics for large projects using linters, but they are not statistically significant.

RQ 2: Interpretation of the Results

Despite our clear result of superior code quality in projects using linters, most projects are far from what we would consider high-quality projects. We assume that a lot of projects integrate linters but do not enforce the resolvement of their findings. Linters are, therefore, considerably used to give developers suggestions for possible code quality improvements. Developers, on the other hand, might often ignore these warnings. This could be due to various reasons, such as the (high) time required for remediation, difficulties in removing the warnings, or the fact that the removal is considered unnecessary.

Another possible explanation is that a majority of projects in which we detect linter integration with the LinterDetectionTool do not actually automatically show warnings in PR reviews or cause failure or warnings in the build output. Therefore the tool might stay unnoticed by most developers and isn't used in the development process consistently.

4.3 RQ 3: Impact of linters on software quality

This research question aims at providing insights regarding the correlation between linter usage and the popularity of projects on GitHub.

The interpretation of Tables and Figures is equivalent to *RQ 2*.

H 9: *GitHub-stars*

This hypothesis investigates if we can observe significant differences in stars among projects using and not using linters.

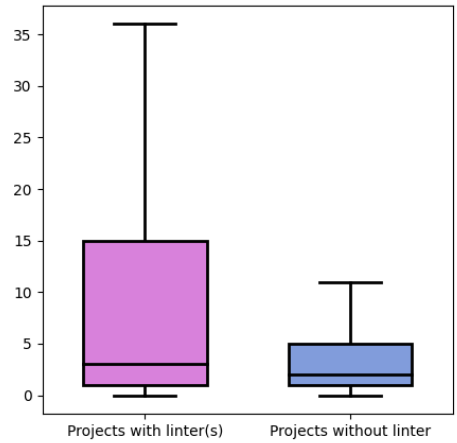
Results for *H 9*

Figure 4.8 clearly shows that projects that use linters, as testified by the Wilcoxon tests (Table 4.8), have more stars on GitHub than projects that do not use linters.

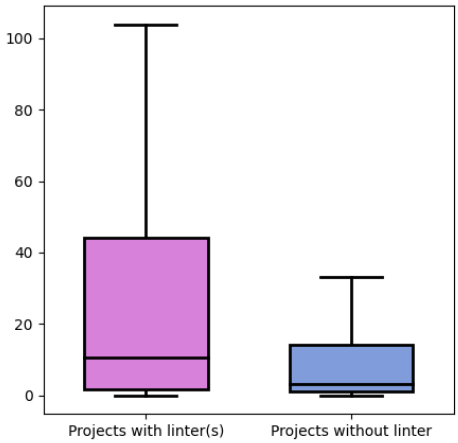
Within all three groups, the project with the maximum number of stars is a lot higher among projects using linters compared to projects without linter. While the medians of small projects only differ by one star, the most popular small, medium, and large projects with linter(s) have 2,423, 3,214, and 1,817 stars. The most popular project without linter has 1,371 stars. Also, the median of 2 among all projects without linters is 5 stars lower than the median for projects using linters.

We conclude:

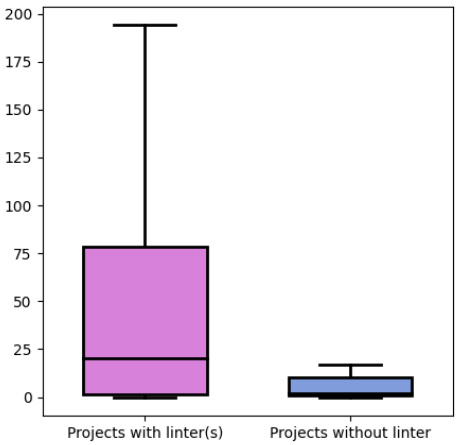
Projects using linters have *significantly* more GitHub stars.



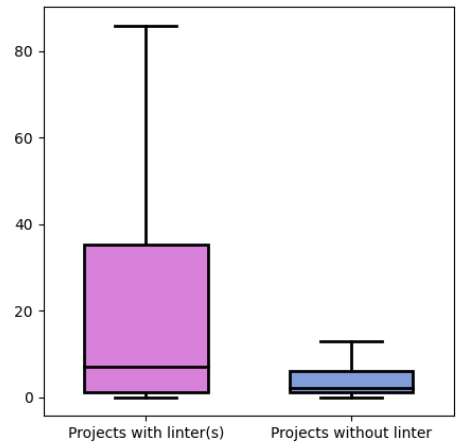
(a) small



(b) medium



(c) large



(d) all

Figure 4.15: Comparison of GitHub-stars without outliers

Set Name	Set Size	Avg. KSLOC	Mean	Median	p-value
Projects with linter(s) (small)	313	5	30.74	3	$9 \cdot 10^{-8}$
Projects without linter (small)	2828	4	9.7	2	
Projects with linter(s) (medium)	392	38	64	10	$8 \cdot 10^{-13}$
Projects without linter (medium)	1200	27	17.9	3	
Projects with linter(s) (large)	131	202	81.8	20	$7 \cdot 10^{-6}$
Projects without linter (large)	77	213	12	2	
Projects with linter(s) (all)	836	51	54.3	7	$1 \cdot 10^{-36}$
Projects without linter (all)	4105	15	12.1	2	

Table 4.8: Comparison of GitHub-stars (Numbers accompanying Figure 4.15 including outliers)

Interpretation of the Results of *H 9* and therefore *RQ 3*

Despite observing that projects using liners have more GitHub stars, we believe that gaining popularity because of linter usage is rarely the case. We assume that most projects that are more popular also have a higher number of contributors. In *H 2*, we already observed that there exists a correlation between linter usage and team size. We, therefore, imagine that while projects gain popularity, the number of contributors increases. With an increasing number of contributors, we expect linter integration as elaborated in the interpretation of *H 2*.

This assumption is to some degree supported by the observation made by Veihelmann in [Vei16] that there exists no correlation between code quality and popularity on GitHub. But if developers would give stars to projects with high code quality, we would expect this. Our projects using linters tend to have better code quality, so we would expect Veihelmanns findings to be similar to ours. Therefore we do generally not believe that the linter is the cause for the success of the project.

On the other hand, our findings again support that development teams consider linters as useful for project maintainability. We do expect popular projects to be of higher importance to the community. Therefore it is especially important to assure the longevity and high maintainability of these projects. The fact that popular projects are more likely to have linter integration(s) shows that the community values the linters and views them as valuable tools for assuring software quality.

5 Threats to Validity

We recognize that there are a few threats to our reported results. Therefore, we explain potential causes for undesired effects on our conclusions' correctness and how we deal with them. In this regard, we distinguish between internal, external, and construct validity.

5.1 Internal validity

The following paragraphs point out to which extent the validity of our conclusions is justifiable.

Accuracy of the LinterDetectionTool

Our analysis results highly rely on the accuracy of the LinterDetectionTool. Despite rigorous manual testing of the tool to improve its accuracy (Section 3.5), there is little guarantee that it correctly identifies repositories in which linters are being used. Since the tool solely relies on detecting linter usage by detecting signal words in files and file paths, it can not detect linters that do not require configurations within the repositories. The possibly most common scenario we are unable to detect is the integration of linters via IDE plugins. Also, the set of linters we are trying to detect might not be exhaustive.

The LinterDetectionTool is implemented in Python and can be found on GitHub[1]. To assure that it fulfills its expected behavior, we wrote tests with *unittest*[2]. To further manifest its abilities in detecting linter usage, we studied and improved it as much as possible within the scope of this study. A detailed explanation and manual evaluation of the LinterDetectionTool was conducted and is documented in Section 3.5.

We could further improve the LinterDetectionTool by adding features like evaluating commit messages for signals to detect linters that do not require integration via the repository. The manual evaluation could have possibly been improved by surveying the developers of the manually inspected repositories if they actually use the linters that are integrated. We could also improve our manual evaluation by surveying the developers of the manually inspected repositories to ask if they use the linters that they have integrated into their projects. With more precise data, it might have been possible to generate a dataset with which we could train the LinterDetectionTool to recognize the usage of linters with the help of machine learning. These improvements would have exceeded the scope of this work.

[1] https://github.com/KorbiQWeidinger/LinterDetectionTool
[2] https://docs.python.org/3/library/unittest.html

Implementation of Procedures and Analysis

Results may be incomplete or inaccurate due to mistakes made during the development or analysis process. In the course of this work, we implemented numerous Python scripts. These scrips were used to find possibly relevant projects through GHTorrent, collect data via the GitHub REST API, clone and analyze projects from GitHub, integrate exclusion mechanisms for test code, generated files and third-party libraries, detect linter usage through the LinterDetectionTool and evaluate the collected data. Due to the complexity emerging from combining all these tasks, we can not completely rule out the possibility of making mistakes throughout the development and evaluation process.

To mitigate the risk of errors, we randomly sanity-checked collected data manually to assure its correctness. To analyse code metrics, we relied on TS, a well-established static analyser. We also compared the results of our analysis to Veihelmanns work [Vei16], which included an analysis of the same code metrics for GitHub projects as our work, to check if we acquire similar metric distributions. We are therefore confident that our findings are trustworthy to a reasonable degree.

Accuracy of GHTorrent and GitHub API

We used GHTorrent to identify and the GitHub API to collect data about our study objects. The validity of our collected data highly depends on the correctness of these sources, and erroneous data could lead to incomplete or meaningless results.

To reduce the risk of collecting outdated data, we solely relied on data acquired through the GitHub API collected at the time of cloning and analyzing the projects. To assure that the collected data is accurate, we randomly compared it to the data visible on the projects GitHub page under *https://github.com/<owner_name>/<project_name>* at the time of retrieval. Since we did not detect signs of potential defects, we estimate the risk of bias within our data sources as very small.

5.2 External validity

The results of this work are based exclusively on software projects written in Java that are available via the GitHub platform. Therefore the conclusions presented in this thesis may only be applicable for Java projects from GitHub. OSS systems' code quality might be different (superior or inferior) to closed source systems. Additionally, there are individual metric differences for different programming languages [Vei16]. Therefore we recommend that our conclusions are essentially only applied to the code quality of Java projects on GitHub. However, it is also possible that our conclusions are correct for a

broader set of software applications, especially when taking other OSS Java projects into account.

5.3 Construct validity

This section discusses whether our case study is suitable to answer the research questions proposed.

Measuring Software Quality

In this study, we examine software quality through various code metrics to evaluate the effectiveness of static analysis tools in OSS systems. However, different studies use different approaches to measure software quality.

The ISO/IEC has tried to define evaluation methods for the quality of software products to define common standards, called the SQuaRE (Systems and software Quality Requirements and Evaluation) series [Nak+16]. In practice, there are two different types of quality models (QMs) [Wag+12]. On the one side, specifications like the ISO 25010 describe and structure general concepts that constitute high-quality software. Those generalized concepts are challenging to use for actual quality assessments [Wag+12]. On the other side, some QMs focus on architectural paradigms or single aspects of software quality (e.g., maintainability) [Wag+12]. Examples would be using the SEATC code quality index [Dra96] that combines a set of code metrics or the analysis of bug reports [Aye+08].

Since many other studies consider code quality, in particular code metrics, to evaluate software quality [Vei16; Sch+15; Wag+12], we are confident that the methodology presented in this work concedes us to draw conclusions and that code quality is a worthy subject of scientific study.

Expectation of Researchers

In the course of this work, we introduce and evaluate the validity of nine hypotheses. To some degree, our thoughts related to these assumptions are biased due to our expectations and personal experience. Therefore it is not unlikely that we investigated questions that are not very relevant in practice. Additionally, our scope was limited to mentioned hypotheses, while other aspects could have further been analyzed.

Experimental Design

We used static code analysis toolkit TS for calculating desired code metrics of our study objects. While TS is a well-established tool for such purposes, the particular settings of our analysis (e.g., a minimal clone length of 10) might not represent meaningful values that are useful in practice. Furthermore, many of our chosen thresholds are, to some extent, arbitrary, which implies that different preferences may lead to results differing from ours. To mitigate this threat, we used two different thresholds for many assessments (e.g., 30 and 50 statements for method length violations). Despite adding this differentiation, we cannot eliminate that these values might reflect unrealistic expectations about real-world software development.

We tried to express our reasoning behind each threshold as well as possible. Existing studies use similar concepts and code metric values to judge software quality which supports our propositions [Vei16; Sch+15].

6 Conclusion and Outlook

In this chapter, we summarize the findings of this thesis and discuss opportunities for future research related to the topic of this thesis.

6.1 Wrap-up

In this thesis, we presented a large-scale case study on GitHub projects written in Java. We analyzed their source code via static analysis as an indicator for overall software quality. We downloaded and analysed 5,911 individual repositories while putting specific project selection criteria into practice. We additionally systematically collected information like team size and GitHub stars for each project.

We selected a set of code metrics, namely clone coverage, comment completeness, nesting depth, method length, and file size, to compare the software quality of different systems. We reused exclusion mechanisms introduced by Veihelmann in [Vei16] to exclude generated files and test code from the analysis process. To answer our problem statement and collect additional insights, we formulated nine hypotheses about the relationship between code quality, linter usage, team size, project size, and project popularity.

In order to detect linter usage in projects, we had to develop a tool which we named LinterDetectionTool. The tool detects the linters Checkstyle, FindBugs/SpotBugs, PMD, SonarQube, CodeQL, Codacy, and Coverity by searching a project for signal words and signal file names extracted from their documentations. To make the tool as accurate as possible, we manually analysed signal findings for around 50 GitHub projects for each linter.

We acquired a set of 606,010 potential projects via the GHTorrent data set while applying specific selection criteria. To make the LinterDetectionTool as accurate as possible we manually analysed signal findings for around 50 GitHub projects for each linter by randomly analysing projects from the collected data. We afterward randomly selected 5,911 of these projects for analysis.

We collected their newest data via the GitHub API for our study objects, verified it matched additionally specified selection criteria, and finally downloaded and analysed the projects with TS and the LinterDetectionTool. After applying our post-analysis criteria, we were left with a total of 5,017 study objects. We stored the overall results to allow follow-up investigations of the repositories' quality metrics and information about the usage of static analysis tools. Based on the collected data, we then evaluated the hypotheses mentioned above using statistical methods.

We could show that projects using linters have significantly better code quality concerning our selected code metrics. We also observed that linters are more likely to be used in projects with larger teams. Furthermore, we could make out that there seems to be a correlation between linter usage and the popularity of projects. Nonetheless, our research shows that projects using linters, while having *better* code quality in general, do not necessarily result in codebases that we would consider *high-quality*.

6.2 Future work

There are numerous possible starting points for future research to verify, improve and extend the results of our approach. In a nutshell, we see four main fields for such future works: enhancements of the analysis process, different project selection, and applying different approaches for measuring software quality.

There are various ways to improve the analysis process. One of the most significant components of our analysis is the LinterDetectionTool. The tool could be improved in numerous ways. We see numerous improvement possibilities. These include extending and further improving the list of signals. We could also add missing SATs, including linters for languages besides Java, to allow examinations for other programming languages. Also, we could add additional features to detect linters whose integration is not visible via file content of file names; such implementations could include identifying linters through the analysis of commits, issues, or pull requests. Furthermore, the patterns used for identification test code and generated files could be enhanced to improve their exclusions and thus further reduce the likelihood of biased analysis results.

Secondly, we needed to limit our scope to the language Java and the platform GitHub. On the one side, we could expand our analysis to include other popular languages. This would allow comparisons between the impact of linters in different programming languages. On the other side, we could additionally analyse projects from different platforms or closed source projects. This would allow us to explore the differences in the impact of static analysis tools between OSS platforms and closed-source projects.

Lastly, future works could investigate the impact of static analysers on different quality aspects. Besides extending the list of code metrics, one could also measure the effect of linters on other quality factors. Other studies have already investigated aspects like reducing human effort in peer code reviews by using automatic static analysis [Bal13]. There are also studies using static analysis to find and reduce bugs [Aye+08]. Nagappan, for example, analysed if the number of SAT findings could is an early indicator for pre-release defect density [NB05]. Others conducted surveys to evaluate why developers use static analysis tools [NWA20; Joh+13]. Also, a paper exists investigating if developers

fix warnings created by linters, mining issue data, and surveying developers [Mar+19]. Since most of these papers rely on surveys or small data sets, their findings could be further extended by automating and upscaling their analysis to large-scale studies. An exciting aspect could also be the correlation between linter usage and usability. This could be investigated by evaluating user satisfaction via surveys. The results of this type of research could help further assess whether the use of SATs plays an essential factor in developing a successful software system.

Acronyms

LOC lines of code

SLOC source lines of code

KLOC thousand LOC

KSLOC thousand SLOC

ConQAT Continuous Quality Assessment Toolkit

TS Teamscale

OSS open source software

CLI command-line interface

CI Continuous Integration

LGPL GNU Lesser General Public License

PR Pull Request

VSC Visual Studio Code

BSD Berkeley Software Distribution

CD Continuous Delivery

CC Cyclomatic Complexity

SAT static analysis tool

RegExp regular expression

List of Figures

List of Tables

Appendix

Linter Detection Analysis Details

RegExp Whitelist for File Names

The LinterDetectionTool only searches files for signals if the filename contains one of the following regular expressions.

Filenames:

```
JENKINS
Jenkinsfile
Change
CHANGE
change
LICENSE
.cfg
.java
.xml
.xsl
.md
.gitignore
.sh
.gradle
.yml
.adoc
BUILDING.txt
.properties
.fbprefs
.settings
.pom
.lockfile
.adoc
.toml
```

RegExp Blacklist for File Names

This list of signals may indicate linter integration, but for linters not considered in this study.

Filenames:

```
.deepsource.toml
```

File Content:

```
semgrep
/infer
DeepSource
static␣analysis␣tool
pvs-studio-analyzer
returntocorp/semgrep
infer-maven-plugin
com.uber:infer-plugin
#mooseDisplayString
infer/
LDRA␣Testbed
visual␣expert
kiuwan
ConQAT
reshiftsecurity
fortify.ps.maven.plugin
INFER_VERSION
␣reshift
infer-linux
Parasoft
moosequery
pvsstudio
visual-expert
␣fortify
Jenkins-Android-Infer
␣embold
MooseModel
com.pvsstudio
SHORT_VERSION_PVS
```

```
Fortify
mooseModel
deepsource
www.conqat.org
SEMGREP_RULESET
pvsstudio-cores
PVS_STUDIO_CORE
PVSStudio
Kiuwan
MooseEntity
hfcca
com.facebook.infer
com.facebook.infer.annotation
Visual␣Expert
Static␣Analysis␣Tool
mooseName
semgrep-action
RIPSTECH
moosetechnology
jarchitect
infer-annotation
␣RIPS
mooseDescription
VISUAL␣EXPERT
␣Embold
MooseDescription
veracode
Veracode
com.uber.infer.java
moosetechnology/Moose
infer␣run
inferPlugin
conqat
pvsAnalyze
JArchitect
semgrep-agent
Jtest
Semgrep
PVS-Studio-Java
```

```
RIPS-TECH
␣Reshift
codescene
com.pvsstudio.PvsStudioGradlePlugin
moose.changes
CodeScene
infer␣capture
PVS-Studio
pvs-studio
Codescene
Hfcca
pvsstudio-maven-plugin
moose.image
com.uber.infer.android
```

Linter Signals

This section lists all the RegExps used in order to detect the individual linters. For each configured linter, the tool searches for the following file names and contents.

Checkstyle

Filenames:

```
google_checks.xml
checkstyle-suppressions.xml
sun_checks.xml
.*checkstyle.*
checkstyle.xml
checkstyle_config.xml
linter.yml
maven_checks.xml
```

File Content:

```
super-linter
Check␣Style
checkstyleMain
```

```
check␣style
checkstyleTest
checkstyle-action
mega-linter
checkstyleSourceSet
Super-Linter
CheckStyle
megalinter
Mega-Linter
checkstyle:checkstyle
checkstyle:checkstyle-aggregate
checkstyle:check
superlinter
codacy-checkstyle
Run␣check␣style
checkstyle
maven-checkstyle-plugin
Checkstyle
checkstyleVersion
```

FindBugs

Filenames:

```
fb-contrib-.*.jar
findbugs-exclude.xml
.*findbugs.*
.fbprefs
```

File Content:

```
findbugsPlugins
Findbugs
configurations.findbugsPlugins.dependencies
com.mebigfatguy.fb-contrib
findbugsVersion
findbugs
com.mebigfatguy.fb-contrib:fb-contrib
project.configurations.findbugsPlugins
```

```
FINDBUGS
com.google.code.findbugs
FindBugs
findbugs-maven-plugin
fb-contrib
```

SpotBugs

Filenames:

```
spotbugs-exclude.xml
.*spotbugs.*
spotbugs-ant.jar
sb-contrib-.*.jar
```

File Content:

```
spotbugs
com.github.spotbugs
spotbugs-maven-plugin
Find␣Security␣Bugs
SpotBugs
Spotbugs
findsecbugs
com.mebigfatguy.sb-contrib
sb-contrib
```

PMD

Filenames:

```
pmd.*.xml
```

File Content:

```
pmdVersionCPDTask
"pmd"
codacy-pmd
copy-paste␣detector
```

```
CPD-START
pmdSourceSet
␣PMD
net.sourceforge.pmd
pmd-java
pmd-core
pmdMain
'pmd'
pmdTest
maven-pmd-plugin
CPD-END
codacy-pmdjava
Copy/Paste␣Detector
```

CodeQL

Filenames:

```
.*codeql.*
lgtm-cluster-config.yml
codeql-analysis.yml
lgtm-config-gen.jar
lgtm.yml
```

File Content:

```
LGTM_CREDENTIALS_PASSWORD
CodeQL
lgtm-down
lgtm-controller
lgtm-status
codeql
LGTM_RAM
lgtm-upgrade
lgtm-workerhost
lgtm-import-ssl-certificate
lgtm-build-requirements
lgtm-cli
LGTM_THREADS
```

SonarQube

Filenames:

```
sonar-project.properties
sonar-generic-coverage.xsd
```

File Content:

```
sonar.host.url
sonar.*jacoco
sonar.qualitygate
sonar.scanner
SonarLint
SONARQUBE
Sonarqube
SonarJava
sonar.cpd
sonarlint
sonar.junit.reportPaths
sonar.cluster
sonar.verbose
sonar.links.ci
sonarqube
sonarsource
sonar.projectName
sonar.sources
sonar.ws.timeout
sonar.coverageReportPaths
sonar-maven-plugin
sonar.login
sonar.java.jdkHome
sonar.projectVersion
sonar-scanner-maven
sonar.projectKey
sonar.analysis
SonarCloud
sonar.scm.exclusions.disabled
SonarQube
SonarSource
sonar.password
```

```
sonar.links.scm
sonar.coverage.jacoco.xmlReportPaths
sonarcloud
sonar.working.directory
sonar.tests
sonar.externalIssuesReportPaths
sonar.projectDescription
sonar.links.issue
sonar.log
```

Coverity

File Content:

```
coverity
Coverity
```

Codacy

File Content:

```
Codacy
codacy
```

Code Exclusion Patterns

Path-based Exclusions

The following set of paths was excluded from our TS analysis.

```
**/lib/**
**/libs/**
**/lib-src/**
**/libs-src/**
**/Lib/**
**/Libs/**
**/generated-src/**
**/libraries/**
**obfuscated**
**/snippet/**
**/snippets/**
**/sample/**
**/samples/**
**/example/**
**/examples/**
**/demo/**
**/ext/**
**/testing/**
**/testsuite/**
**/test/**
**/test-data/**
**/test_data/**
**/tests/**
**/build/**
**/src-gen/**
**/thirdparty/**
**/3rdparty/**
**/third-party/**
**/third_party/**
**/framework/**
**/frameworks/**
**/src-min/**
```

```
**/include/**
**/Include/**
**/plugins/**
**/external/**
**/external-src/**
**/intermediates/**
**/bin/**
**/packages/**
**/obj/**
**/tags/**
**/branches/**
**/res/**
**/jdk/**
**/modules/**
**/package-info.java
**/module-info.java
**/R.java
```

Test Framework Checks

We detected test frameworks JUnit and TestNG and excluded test files with the following content from our TS analysis.

```
import␣.*junit.*
import␣.*testng.*
```

Bibliography

[Abt20] D. Abts. “Javadoc.” In: *Grundkurs JAVA*. Springer, 2020, pp. 217–220.

[Aye+08] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. “Using static analysis to find bugs.” In: *IEEE software* 25.5 (2008), pp. 22–29.

[Bal13] V. Balachandran. “Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation.” In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 931–940. DOI: 10.1109/ICSE.2013.6606642.

[Bau+12] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt. “A framework for incremental quality analysis of large software systems.” In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2012, pp. 537–546.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. “Quantitative evaluation of software quality.” In: *Proceedings of the 2nd international conference on Software engineering*. 1976, pp. 592–605.

[BBS13] A. Begel, J. Bosch, and M.-A. Storey. “Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder.” In: *IEEE software* 30.1 (2013), pp. 52–66.

[BC14] A. Bosu and J. C. Carver. “Impact of developer reputation on code review outcomes in oss projects: An empirical investigation.” In: *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*. 2014, pp. 1–10.

[BM14] S. Bhatia and J. Malhotra. “A survey on impact of lines of code on software complexity.” In: *2014 International Conference on Advances in Engineering & Technology Research (ICAETR-2014)*. IEEE. 2014, pp. 1–4.

[Bor+15] H. Borges, M. T. Valente, A. Hora, and J. Coelho. “On the popularity of GitHub applications: A preliminary note.” In: *arXiv preprint arXiv:1507.00604* (2015).

[BV18] H. Borges and M. T. Valente. “What’s in a github star? understanding repository starring practices in a social coding platform.” In: *Journal of Systems and Software* 146 (2018), pp. 112–129.

[CC17] J. Creswell and J. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2017. ISBN: 9781506386713.

[CD11] C. Calcagno and D. Distefano. “Infer: An Automatic Program Verifier for Memory Safety of C Programs.” In: *NASA Formal Methods*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465. ISBN: 978-3-642-20398-5.

[Che] Checkstyle. URL: https://checkstyle.org/ (visited on 01/05/2021).

[coda] codacy. URL: https://www.codacy.com/ (visited on 05/03/2021).

[codb] codacy. URL: https://docs.codacy.com/ (visited on 07/13/2021).

[Con15] L. M. Connelly. “Research questions and hypotheses.” In: *Medsurg Nursing* 24.6 (2015), pp. 435–436.

[Cor14] J. E. Corbly. “The Free Software Alternative: Freeware, Open Source Software, and Libraries.” In: *Information Technology and Libraries* 33.3 (Sept. 2014), pp. 65–75. DOI: 10.6017/ital.v33i3.5105.

[DB15] M. Demarne and E. Burmako. *Style Checking With Scala. Meta*. Tech. rep. 2015.

[Dei+08] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y. Parareda, and M. Pizka. “Tool Support for Continuous Quality Control.” In: *IEEE Software* 25.5 (2008), pp. 60–67. DOI: 10.1109/MS.2008.129.

[Dra96] T. Drake. “Measuring software quality: a case study.” In: *Computer* 29.11 (1996), pp. 78–87. DOI: 10.1109/2.544241.

[Fac] Facebook. URL: https://fbinfer.com/ (visited on 05/17/2021).

[Fah+16] L. Fahrmeir, C. Heumann, R. Künstler, I. Pigeot, and G. Tutz. *Statistik: Der weg zur datenanalyse*. Springer-Verlag, 2016.

[Fak+19] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova. “Improving source code readability: theory and practice.” In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 2–12.

[Fin] FindBugs. URL: http://findbugs.sourceforge.net/index.html (visited on 02/05/2021).

[FN00] N. E. Fenton and M. Neil. “Software Metrics: Roadmap.” In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE ’00. Limerick, Ireland: Association for Computing Machinery, 2000, pp. 357–370. ISBN: 1581132530. DOI: 10.1145/336512.336588.

[GHJ12] N. Göde, B. Hummel, and E. Juergens. “What clone coverage can tell.” In: *2012 6th International Workshop on Software Clones (IWSC)*. 2012, pp. 90–91. DOI: 10.1109/IWSC.2012.6227880.

[Gita] GitHub. URL: https://codeql.github.com/docs/ (visited on 07/13/2021).

[Gitb] GitHub. URL: https://docs.github.com/en (visited on 05/17/2021).

[Göd+14] N. Göde, L. Heinemann, B. Hummel, and D. Steidl. "Qualität in Echtzeit mit Teamscale." In: *Softwaretechnik-Trends* 34.2 (2014).

[Gou13] G. Gousios. "The GHTorrent dataset and tool suite." In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. Best data showcase paper award. San Francisco, CA, USA, May 2013, pp. 233–236. ISBN: 978-1-4673-2936-1.

[HHS14] L. Heinemann, B. Hummel, and D. Steidl. "Teamscale: Software Quality Control in Real-Time." In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 592–595. ISBN: 9781450327688. DOI: 10.1145/2591062.2591068.

[HKV07] I. Heitlager, T. Kuipers, and J. Visser. "A Practical Model for Measuring Maintainability." In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8.

[Hum] D. B. Hummel. URL: https://www.cqse.eu/en/news/blog/conqat-end-of-life/ (visited on 05/18/2021).

[HWY09] T. Honglei, S. Wei, and Z. Yanan. "The Research on Software Metrics and Software Complexity Metrics." In: *2009 International Forum on Computer Science-Technology and Applications*. Vol. 1. 2009, pp. 131–136. DOI: 10.1109/IFCSTA.2009.39.

[JDH10] E. Juergens, F. Deissenboeck, and B. Hummel. "Code similarities beyond copy & paste." In: *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE. 2010, pp. 78–87.

[Joh+13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[Jue+09] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. "Do code clones matter?" In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 485–495.

[Jue11] E. Juergens. "Why and how to control cloning in software artifacts." PhD thesis. Technische Universität München, 2011.

[Kal+14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. “The promises and perils of mining github.” In: *Proceedings of the 11th working conference on mining software repositories*. 2014, pp. 92–101.

[KAY14] M. Kapdan, M. Aktas, and M. Yigit. “On the structural code clone detection problem: a survey and software metric based approach.” In: *International Conference on Computational Science and Its Applications*. Springer. 2014, pp. 492–507.

[Kim+05] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. “An Empirical Study of Code Clone Genealogies.” In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 187–196. ISSN: 0163-5948. DOI: 10.1145/1095430.1081737.

[KWR10] N. Khamis, R. Witte, and J. Rilling. “Automatic quality assessment of source code comments: the JavadocMiner.” In: *International Conference on Application of Natural Language to Information Systems*. Springer. 2010, pp. 68–79.

[Lie83] B. P. Lientz. “Issues in Software Maintenance.” In: *ACM Comput. Surv.* 15.3 (Sept. 1983), pp. 271–278. ISSN: 0360-0300. DOI: 10.1145/356914.356919.

[Mar+19] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto. “Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube.” In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 209–219. DOI: 10.1109/ICPC.2019.00040.

[Mar08] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.

[McC76] T. J. McCabe. “A complexity measure.” In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[Mooa] O. de Moor. URL: https://blog.semmle.com/secure-software-github-semmle/ (visited on 07/13/2021).

[Moob] Moosetechnology. URL: http://moosetechnology.org/ (visited on 05/17/2021).

[Mooc] Moosetechnology. URL: http://themoosebook.org/book (visited on 05/18/2021).

[Nak+16] H. Nakai, N. Tsuda, K. Honda, H. Washizaki, and Y. Fukazawa. “A SQuaRE-based software quality evaluation framework and its case study.” In: *2016 IEEE Region 10 Conference (TENCON)*. 2016, pp. 3704–3707. DOI: 10.1109/TENCON.2016.7848750.

[NB05] N. Nagappan and T. Ball. “Static analysis tools as early indicators of pre-release defect density.” In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 2005, pp. 580–586. DOI: 10.1109/ICSE.2005.1553604.

[NWA20] L. Nguyen Quang Do, J. Wright, and K. Ali. “Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations.” In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1. DOI: 10.1109/TSE.2020.3004525.

[PMD] PMD. URL: https://pmd.github.io/latest/ (visited on 05/03/2021).

[RT17] T. V. Ribeiro and G. Travassos. “Who is Right? Evaluating Empirical Contradictions in the Readability and Comprehensibility of Source Code.” In: *CIbSE*. 2017, pp. 595–608.

[SAO05] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. “A study of the documentation essential to software maintenance.” In: *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. 2005, pp. 68–75.

[Sch+15] K. Schüler, R. Trogus, M. Feilkas, and T. Kinnen. “Managing product quality in complex software development projects.” In: *Proceedings of the Embedded World Conference*. Citeseer. 2015.

[SDZ07] D. Schreck, V. Dallmeier, and T. Zimmermann. “How Documentation Evolves over Time.” In: *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*. IWPSE ’07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 4–10. ISBN: 9781595937223. DOI: 10.1145/1294948.1294952.

[Sema] Semgrep. URL: https://semgrep.dev/docs/ (visited on 05/03/2021).

[Semb] Semmle. URL: https://lgtm.com/ (visited on 07/13/2021).

[SHJ13] D. Steidl, B. Hummel, and E. Juergens. “Quality analysis of source code comments.” In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 83–92. DOI: 10.1109/ICPC.2013.6613836.

[SHJ14] D. Steidl, B. Hummel, and E. Juergens. “Incremental origin analysis of source code files.” In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 42–51.

[SK21] M. Steinbeck and R. Koschke. “Javadoc Violations and Their Evolution in Open-Source Software.” In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021, pp. 249–259. DOI: 10.1109/SANER50967.2021.00031.

[Son] SonarSource. URL: https://www.sonarqube.org/downloads/ (visited on 05/17/2021).

[Spo] SpotBugs. URL: https://spotbugs.github.io/ (visited on 05/02/2021).

[Syn] Synopsys. URL: https://scan.coverity.com/ (visited on 07/12/2021).

[Vei16] D. Veihelmann. "How does the Programming Language affect Software Quality in Open-Source Software?" MA thesis. Technische Universität München, 2016.

[Wag+12] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidi, A. Goeb, and J. Streit. "The Quamoco product quality modelling and assessment approach." In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 1133–1142. DOI: 10.1109/ICSE.2012.6227106.

[Wah+04] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. "Clone detection in source code by frequent itemset techniques." In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. 2004, pp. 128–135. DOI: 10.1109/SCAM.2004.6.

[WKK07] D. Wilking, U. F. Kahn, and S. Kowalewski. "An Empirical Evaluation of Refactoring." In: *e Informatica Softw. Eng. J.* 1.1 (2007), pp. 27–42.