Some SonarQube Issues have a Significant but Small Effect on Faults and Changes. A Large-Scale Empirical Study

Valentina Lenarduzzi[a], Nyyti Saarimäki[b], Davide Taibi[b]

[a]*LUT University, Lahti (Finland)*
[b]*Tampere University, Tampere (Finland)*

Abstract

Context. Companies frequently invest effort to remove technical issues believed to impact software qualities, such as removing anti-patterns or coding styles violations.
Objective. We aim to analyze the diffuseness of SonarQube issues in software systems and to assess their impact on code changes and fault-proneness, considering also their different types and severities.
Method. We conducted a case study among 33 Java projects from the Apache Software Foundation repository.
Results. We analyzed 726 commits containing 27K faults and 12M changes in Java files. The projects violated 173 SonarQube rules generating more than 95K SonarQube issues in more than 200K classes. Classes not affected by SonarQube issues are less change-prone than affected ones, but the difference between the groups is small. Non-affected classes are slightly more change-prone than classes affected by SonarQube issues of type Code Smell or Security Vulnerability. As for fault-proneness, there is no difference between non-affected and affected classes. Moreover, we found incongruities in the type and severity assigned by SonarQube.
Conclusions. Our result can be useful for practitioners to understand which SonarQube issues should be refactored and for researchers to bridge the missing gaps. Moreover, results can also support companies and tool vendors in identifying SonarQube issues as accurately as possible.

Keywords: Change-proneness, Fault-proneness, SonarQube, Empirical Study

Email addresses: `valentina.lenarduzzi@lut.fi` (Valentina Lenarduzzi), `nyyti.saarimaki@tuni.fi` (Nyyti Saarimäki), `davide.taibi@tuni.fi` (Davide Taibi)

1. Introduction

Technical Debt (TD) is a metaphor from the economic domain that refers to different software maintenance activities that are postponed in favor of the development of new features in order to get short-term payoff [1]. Just as in the case of financial debt, the additional cost will be paid later.

The growth of TD commonly slows down the development process [1][2]. Therefore, software companies need to control it by managing the many factors that make its evolution often unpredictable, such as internal or external forces affecting the business or its environment [3][4]. In practice, these factors can include technical issues and information that can be derived from the source code or the software process, such as the usage of specific patterns, compliance with coding or documentation conventions, and architectural issues. If such issues are not fixed, they generate TD.

Different types of TD exist: requirements debt, code debt, architectural debt, design debt, test debt, build debt, documentation debt, infrastructure debt, versioning debt, and defect debt [2]. Some types of TD, such as code TD, can be measured using static analysis tools, which is why companies have started to adopt code TD analysis tools like SonarQube, Cast, and Coverity Scan.

SonarQube is one of the most frequently used open-source code TD analysis tools [5], having been adopted by more than 120K users[1], including more than 100K public open-source projects[2]. SonarQube allows for code TD management by monitoring the evolution of TD and alerting developers if selected issues increase beyond a specified threshold or, even worse, grow out of control. TD monitoring can also be used to support the prioritization of repayment actions where issues are resolved (e.g., through refactoring) [6][7].

SonarQube monitors the TD by analyzing code compliance against a set of rules. If the code violates a rule, SonarQube adds the time needed to refactor the violated rule as part of the technical debt, thereby creating an issue. SonarQube classifies issues into three main categories: *Code Smells*, i.e., issues that increase change-proneness and the related maintenance effort; *Bugs*, i.e., issues that will result in a fault; and *Security Vulnerabilities*[3]. SonarQube also assigns one of the following five *severity* levels to each rule:

[1]https://www.sonarqube.org
[2]https://sonarcloud.io/explore/projects
[3]SonarQube Rules: https://docs.sonarqube.org/display/SONAR/Rules

Blocker, *Critical*, *Major*, *Minor*, and *Info*. Rules with a higher severity level have a higher impact on the system. As an example, a Sonar issue labeled as of severity *Blocker* and type *Bug* highlights a piece of code that *"has a high probability to cause the application to crash or to corrupt the stored data"*[3]. It is important to note that the term "code smell" adopted in SonarQube does not refer to the commonly known term code smells defined by Fowler et al. [8]. To avoid misunderstanding, in the remainder of this work we refer to any possible issue in the code with the term "*issue*". Moreover, we refer to all the issues detected by SonarQube with the term *"Sonar issues"* and to the SonarQube issues classified as "code smells" with the term *Code Smell*. To avoid misunderstanding with the code smells proposed by Fowler et al., we refer to them as "Fowler's code smells".

The research community has been studying the impact of different types of quality issues that can be injected in the code, with a special focus on Fowler's code smells. Different aspects of these issues such as the equivalence or the similarity of different types of "smells" (e.g. comparing Fowler's code smells and architectural smells [9]), and the diffuseness of Fowler's code smells and SonarQube issues [7][10][11] have been investigated. Moreover, researchers investigated if Fowler's code smells are relevant for developers [12][13][14], how they evolve [15][16][17][18][19], and their impact on software qualities [20], such as change-proneness [10][21][22][23], fault-proneness [7] [10][24][25][26], and maintenance effort [27][28]. While the aforementioned works reported significant results on Fowler's code smells and their change-proneness or fault-proneness, the empirical evidence provided so far on the impact on Sonar issues and code change-proneness and fault-proneness is still limited because of the small number and size of the previous studies on SonarQube issues, the lack of analysis on the magnitude of the observed phenomenon, the lack of within-artifact analysis, and the lack or temporal relation analysis between the presence of smells and the introduction of faults.

Even if SonarQube is used by more than 120K users[2], the impact of *Bugs* on fault-proneness and *Code Smells* on change-proneness have been investigated only by a few works, and only considering change-proneness or fault-proneness. Moreover, these studies were conducted on different datasets, therefore not allowing to directly compare the obtained results.

As for the limited size of previous studies, the study by Tollin et al. [23] analyzed only the change-proneness on two industrial projects. A preliminary study conducted by Lenarduzzi et al. [29] focused only on fault-proneness of 20 open-source projects without considering change-proneness. These works indicated that some Sonar issues, and in particular some Fowler's code smells can be more harmful than others, but their analyses did not take into account the magnitude of the observed phenomenon. Studies that analyzed Sonar issues, did not investigate the aspect in-depth nor considered the impact of the number of occurrences of the same type of issues. As an example, the impact of a Sonar issue considered harmful might be affected by the number of occurrences in the code.

In addition, several studies indicate that classes affected by certain types of issues, in particular Fowler's code smells, are more likely to exhibit defects (or to undergo changes) than other classes [10][21][22]. However, no study has investigated the extent of the phenomenon, that is compare different classes affected with Fowler's code smells to non-affected classes with respect to the average number of exhibited defects.

Researchers have not investigated within-artifact effect [19]. For such classes having a high change-proneness and/or fault-proneness due to the implementation of complex features, it is fundamental to keep track of their change-proneness and fault-proneness during their evolution to better understand the cause (presence of issues) with the possible effect (change-proneness or fault-proneness). This is because in such classes, issues can easily be introduced throughout their lifetime [19], especially in the case where refactoring occurs and issues are removed [30].

Another aspect overlooked in the current research is the temporal relationship between the presence of smells and the introduction of a fault. While there are results suggesting a correlation between the presence of different types of issues (mainly Fowler's code smells) and high fault-proneness and change-proneness, it is still unknown if one causes the other.

In our previous work [7], we investigated the diffuseness of Sonar issues on the same data set adopted in this study. We observed a very high frequency of Sonar issues in each commit of the selected projects. This result validates the expansion of the investigation of the impact of Sonar issues to change-proneness and fault-proneness. If the magnitude of the phenomenon was small - i.e. Sonar issues were poorly diffused, - then the investigation of their impact on change-proneness and fault-proneness would not have been worthwhile.

Therefore, in this paper, we aim at corroborating previous empirical research by investigating their effect on change-proneness and fault-proneness on a large set of software projects.

Palomba et al. [10] have already conducted a study investigating the effect of Fowler's code smells with respect to change-proneness and fault-proneness of code. In their paper, they used commonly adopted data-collection methods and data analysis techniques. As a study similar to our idea had already been published, we decided to conduct our study as a differentiated external replication of their study. We investigated the same goals and research questions on Sonar issues instead of on Fowler's code smells.

This study was conducted using 33 open-source systems by taking a commit from them once every six months, totalling to 726 commits. In the study, we considered the issues detected by SonarQube's default model.

More specifically, the study aims at investigating the impact of Sonar issues classified as *Code Smells* on maintenance properties, focusing specifically on code change- and fault-proneness. We intend to investigate the extent to which previous findings reported by Tollin et al. [23] and Lenarduzzi et al. [29] - obtained on a smaller set of different systems - are confirmed on a larger set of projects.

To the best of our knowledge, this is to date the largest study investigating the relationship between the presence of issues detected by SonarQube, change-proneness and fault-proneness in the same dataset. In addition, the adoption of an existing study design also helped to cope with the limitations of the studies mentioned above by (i) analyzing the fault-proneness magnitude in terms of number of instances of SonarQube violation, and (ii) using the SZZ algorithm [31] to determine whether an artifact was already smelly when a fault was induced. Moreover, in order to enable the replication of our work, the data set used in this study and all the scripts adopted for the analysis are publicly available in our replication package[4].

Structure of the paper. Section 2 describes the study we replicated. Section 3 presents SonarQube, the static analysis tool adopted in this work, while Section 4 presents some related work done by researchers in recent years. In Section 5, we describe the design of our case study, defining the research questions, metrics, and hypotheses, and describing the study context with the data collection and data analysis protocol. In Section 6, we present the achieved results, discussing them in Section 7. In Section 8, we identify the threats to the validity of our study, and in Section 9, we draw conclusions and give an outlook on possible future work.

2. The Replicated Study

In this Section, we summarize the design of the study we replicated [10] and the reasons we conducted our study based on it. The replication was

[4] https://figshare.com/s/240a036f163759b1ec97

performed following the replication guidelines proposed by Carver [32].

2.1. The Original Study

Palomba et al. [10] investigated if the presence of 13 different kinds of Fowler's code smells affect the change-proneness and fault-proneness of classes. They analyzed 395 releases from 30 open-source projects containing 17,350 instances of the investigated code smells. In particular, they investigated:

a The diffuseness of code smells;

b How code smells affect the change-proneness and fault-proneness of classes;

c If the removal of code smells affects the change-proneness and fault-proneness of classes.

Therefore, in addition to change- and fault-proneness, their study investigated the actual diffuseness of the code smells and whether the systems' characteristics and the presence of code smells is correlated using the Spearman correlation.

For studying the change- and fault-proneness they used the non-normalized versions of the definitions we have adopted in this paper (Section 5.4). Calculating change-proneness requires the number of changes between two commits, for that they extracted the change logs from versioning systems and identified the set of classes modified in each commit. Fault-proneness on the other hand requires the number of fixed faults. To obtain the faults, they used the SZZ algorithm [33] for identifying the fault-fixing and fault-inducing commits.

The change-proneness and fault-proneness of the affected and non-affected classes was visually compared by presenting box-plots of the distributions. The similarity of the distributions was confirmed using Mann-Whitney statistical test and Cliff's Delta effect size measure test.

The study concluded that the most diffused smells were related to the complexity and size of the classes. They reported that classes affected by code smells are generally more change-prone and fault-prone than non-affected classes, especially if a class was affected by multiple code smells. They also discovered that removing code smells often reduces change-proneness.

2.2. Motivations

In our previous work [7] we observed a very high frequency of Sonar issues in open-source projects available in the Technical Debt data set [34]. Therefore, the high diffuseness of Sonar issues enabled us to further investigate the impact of Sonar issues on change-proneness and fault-proneness. Our preliminary investigation on the fault-proneness of Sonar issues [29], conducted on a smaller set of projects, highlighted discordant results between the classification assigned by SonarQube and the fault-proneness of each rule. However, that work was conducted with a non-traditional statistical approach by applying seven machine learning models to predict if a Sonar issue is likely to generate a fault in the future.

Based on the previous results, in this work, we aim at analyzing both change-proneness and fault-proneness considering a large set of projects (33 instead of 21) and all the Sonar issues (classified as *Bugs*, *Code Smells*, and *Vulnerabilities*) detectable by SonarQube. Moreover, we aim at investigating the issue using more traditional statistical approaches.

The work published by Palomba et al. [10] already investigated the change-proneness and fault-proneness of different issues (Fowler's code smells). Therefore, as we had similar goals and available data, we decided to adopt the same approach used in Palomba et al.'s work to increase the validity of our work. Our work was designed as a differentiated external replication, focused on Sonar issues instead of Fowler's code smells.

In our previous work [7], we investigated the diffuseness of Sonar issues in the same dataset we use in this work. We observed a very high frequency of Sonar issues in commits and in each Java class of the selected project. Therefore, the high diffuseness of Sonar issues enables us to further investigate the impact of Sonar issues on change-proneness and fault-proneness. If the magnitude of the phenomenon was small - i.e. Sonar issues were poorly diffused, - then the investigation of their impact on change-proneness and fault-proneness would not have been worthwhile.

Fowler's Code smells and SonarQube issues can be both considered as different types of code TD, and therefore Fowler's Code Smells can be considered as different types of Sonar issues. Palomba et al. adopted a common approach to investigate fault-proneness and change-proneness, identifying fault-fixing and fault-inducing commits using the SZZ algorithm [31], Mann-Whitney statistical test, and Cliff's Delta effect size measure test.

3. SonarQube

Nowadays, some of the most popular Automated Static Analysis Tools (ASAT) used by open-source developers are Findbugs, Checkstyle, and SonarQube [35, 36, 5]. Among them, SonarQube is the only one that provides a TD index. It is provided as a service by the sonarcloud.io platform or can be downloaded and executed on a private server. Moreover, SonarQube has been widely adopted by the Apache Software Foundation, starting from September 2019[5].

SonarQube calculates several metrics, such as the number of lines of code and code complexity, and verifies the code's compliance against a specific set of "coding rules" defined for most common development languages. Moreover, it defines a set of thresholds ("quality gates") for each metric and rule. SonarQube generates an issue if the analyzed source code violates a coding rule or a metric is outside a predefined threshold (also named "gate"). The time needed to remove these issues (remediation effort) is used to calculate the remediation cost and technical debt. SonarQube includes reliability, maintainability, and security vulnerabilities. Moreover, SonarQube claims that zero false positives are expected from the Reliability and Maintainability rules[6].

Reliability rules, also named *Bugs*, create issues that "represent something wrong in the code" and that will soon be reflected in a bug. *Code smells* are considered "maintainability-related issues" in the code that decrease code readability and code modifiability. It is important to note that the term "code smell" adopted in SonarQube does not refer to the commonly known term code smells defined by Fowler et al. [8], but to a different set of rules.

SonarQube also classifies the rules into five *severity* levels[7]:

- *Blocker*: *"Bug with a high probability to impact the behavior of the application in production: memory leak, unclosed JDBC connection."* SonarQube recommends immediately reviewing such an issue.

- *Critical*: *"Either a bug with a low probability to impact the behavior of the application in production or an issue which represents a secu-*

[5]SonarQube dashboard for Apache Software Foundation projects https://sonarcloud.io/organizations/apache/projects

[6]SonarQube Rules:https://docs.sonarqube.org/display/SONAR/Rules

[7]SonarQube Issues and Rules Severity:
https://docs.sonarqube.org/display/SONAR/Issues

rity flaw: empty catch block, SQL injection". SonarQube recommends immediately reviewing such an issue.

- *Major*: *"Quality flaw which can highly impact the developer productivity: uncovered piece of code, duplicated blocks, unused parameters"*
- *Minor*: *"Quality flaw which can slightly impact the developer productivity: lines should not be too long, "switch" statements should have at least 3 cases, ..."*
- *Info*: *"Neither a bug nor a quality flaw, just a finding."*

The complete list of violations can be found in the online raw data in our replication package [4].

4. Related Work

In this Section, we report the most relevant works on the diffuseness, change-proneness and fault-proneness of different issues that can be injected in the code.

4.1. Diffuseness of Technical Debt Issues

To the best of our knowledge, the vast majority of publications investigate the distribution and evolution of Fowler's code smells [8] and anti-patterns [37], while a few works have investigated SonarQube violations.

The distribution of Fowler's code smells has been investigated from several angles. A study compared the length a smell is affecting a software system and whether it is refactored [38]. Another study concluded that the overtime distribution of Fowler's code smells does not follow a specific trend [39]. Some of these smells, such as God Class and Shotgun Surgery, do not evolve constantly overtime as they increase during some periods and decrease in others without correlating with project size. Yet, the number of other smells, such as Feature Envy and Long Method, increases constantly overtime [16] [15]. Ranking the diffuseness of Fowler's code smells, Feature Envy, Message Chain, and Middle Man are poorly diffused, while Speculative Generality, Class Data Should Be Private, Inappropriate Intimacy, and God Class are mostly diffused (average 70%) [10]. Close to 80% of the of Fowler's code smells are never removed from the code, and that those code smells that are removed are eliminated by removing the smelly artifact and not as a result of refactoring activities [19].

To the best of our knowledge, only four works have consider code TD calculated by SonarQube [11][40][6][7].

Digkas et al. [11] investigated the evolution of TD over a period of five years using weekly snapshots. The context of the study was 66 open-source software projects from the Apache ecosystem. Moreover, they characterized the lower-level constituent components of TD. The results showed a significant increase in terms of size, the number of issues, and complexity metrics of the analyzed projects. However, they also discovered that normalized TD decreased as the aforementioned project metrics evolved.

Futhermore, in a subsequent study Digkas et al. [6] investigated the accumulation of TD as a result of software maintenance activities. As context, they selected 57 open-source Java software projects from the Apache Software Foundation and analyzed them at the temporal granularity level of weekly snapshots, also focusing on the types of issues being fixed. The study concluded that the largest percentage of TD repayment is created by a small subset of issue types.

Amanatidis et al. [40] investigated the accumulation of TD in PHP applications, focusing on the relationship between debt amount and interest to be paid during corrective maintenance activities. They analyzed ten open-source PHP projects from the perspective of corrective maintenance frequency and corrective maintenance effort related to the amount of interest. They found a positive correlation between the interest and the amount of accumulated TD.

Saarimäki et al. [7] investigated the diffuseness of Sonar issues on the same data set adopted in this work, considering 33 Java projects, reporting that the most frequently introduced Sonar issues are related to low-level coding issues.

4.2. Change-proneness and Fault-proneness of Technical Debt Issues

In the last years, research has been conducted on the change-proneness and fault-proneness of different types of code smells. Recent works have investigated different types of smells, such as code and architectural smells [9], test smells [30] and Fowler's code smells [10], but also the impact of Fowler's code smells on different features such as faults [2][41][42], maintainability [43][44], comprehensibility [45], change frequency [39][46][47][48], change size [39][49], and maintenance effort [50][51]. The results of previous works confirm that classes affected by Fowler's code smells are more fault-prone and change-prone than non-affected classes [21][22][10].

Another investigated aspect is the effect of refactoring on change-proneness and fault-proneness [2][24][26]. They monitored the same classes for a fixed period and comparing the results with non-refactored classes. Results showed that refactoring decreases change-proneness and fault-proneness. It is worth highlighting that this result is also valid on a more generic perspective related to design problems. The result is not limited only to the context of Fowler's code smells [24] or other issues, such as the ones detected by SonarQube. However, research conducted on Fowler's code smells seems to confirm the above results, as some Fowler's code smells (e.g. God Class and Brain Class) are strongly correlated with class errors [2] and class defects [39], while others (e.g. Feature Envy and Shotgun Surgery) do not seem to affect class fault-proneness [25].

Change-proneness was also investigated considering stability and instability as factors [52], and by evaluating how the probability of changes of classes is affected when new functionality is added or when existing functionality is modified [53]. Based on this approach, two different studies [54][55] conducted case studies on change-proneness. They considered the change history as a proxy for the frequency of changes, and the source code structure that affects the probability of a change being propagated in the code [54]. They applied this approach to five open-source projects [55].

Other works investigated the fault-proneness of different types of Fowler's code smells [8], such as MVC smells [56], testing smells [30], Android smells [57], or other metrics (e.g.Pairs of Local Variables with Confusing Names) [58], or proposed different approaches to evaluate fault-proneness considering class structure, changes in class structure, and the class-level history [59].

Only two works investigated the change-proneness and fault-proneness of issues detected by SonarQube [60][23]. Falessi et al. [60] studied the distribution of 16 metrics and 106 SonarQube issues in an industrial project. They applied a *What-if* approach with the goal of investigating what could happen if a specific SonarQube issue had not been introduced in the code and if the number of faulty classes decreases in case the Sonar issue is not introduced. They compared four Machine Learning (ML) techniques (Bagging, BayesNet, J48, and Logistic Regression) on the project and then applied the same techniques to a modified version of the code, where they had manually removed Sonar issues. Their results showed that 20% of the faults could have been avoided if the Sonar issues had been removed.

Tollin et al. [23] used ML to predict the change-proneness of classes based on SonarQube violations and their evolution. They investigated whether SonarQube violations would lead to an increase in the number of changes (code churns) in subsequent commits. The study was applied to two different industrial projects, written in C# and JavaScript. The authors compared the prediction accuracy of Decision Trees, Random Forest, and Naive Bayes.

They report that classes affected by more SonarQube violations have greater change-proneness. However, they did not prioritize or classify the most change-prone violations.

In our previous work [29], we conducted a preliminary investigation on the fault-proneness of SonarQube *Bugs* on a smaller data set. As in this work, we labeled the fault-inducing commits using the SZZ algorithm [33] while we investigated the fault-proneness using seven machine learning models. We found that rules considered as *Bugs* by SonarQube were generally not fault-prone and, consequently, the fault-prediction power of the model proposed by SonarQube is extremely low.

The results of previous works [23][60][7][29] confirm the need for further investigation on the fault-proneness and change-proneness of SonarQube rules. The rules applied by SonarQube for calculating TD should be thoroughly investigated and their harmfulness needs to be further confirmed. Therefore in this work, we went in deep, investigating the fault-proneness and change-proneness of software quality and considered a larger data set of projects than the previous works.

To the best of our knowledge, our work is the first study that investigated SonarQube considering both their change-proneness and fault-proneness on the same set of projects.

5. Case Study Design

We designed our empirical study as a differentiated external replication of the study proposed by Palomba et al. [10]. In this Section, we describe the case study design including the goal and the research questions, the study context, the data collection, and the data analysis procedure. In order to allow our study to be replicated, we have published the complete raw data and the script adopted for the analysis in the replication package[4].

5.1. Goal and Research Questions

The goal of this study was to analyze the diffuseness of Sonar issues in software systems and to assess their impact on the change-proneness and fault-proneness of the code, considering also the type of technical debt issues and their severity.

Accordingly, to meet our expectations, we extended the goal proposed by Palomba et al. [10] investigating Sonar issues instead of Fowler's code smells. Therefore we formulated our goal as follows, using the Goal/Question/Metric (GQM) template [61]:

Purpose	Analyze
Object	technical debt issues
Quality	with respect to their fault-proneness and change-proneness
Viewpoint	from the point of view of developers
Context	in the context of Java projects

Based on the aforementioned goal, we extended Palomba et al. [10] Research Questions (RQs) targeting to fault-proneness and change-proneness of Sonar issues instead of code smells. Therefore, we derived the following RQs:

RQ$_1$ Do classes affected by Sonar issues have a different level of change-proneness and fault-proneness with respect to non-affected ones?

In this RQ, we aim at understanding if classes affected by at least one Sonar issue have a higher chance of being change-prone and fault-prone than non-affected classes measuring the magnitude of the change-proneness and fault-proneness of classes in terms of the number of changes and number of bug fixes. Therefore, we formulated two sub-RQs: RQ$_{1.1}$ to investigate the change-proneness and RQ$_{1.2}$ to investigate the fault-proneness.

Since the presence of SonarQube issues is supposed to decrease the quality of the code, we also expect to find a different level of change-proneness (H$_{1.1.1}$) and fault-proneness (H$_{1.2.1}$) in classes affected by these issues.

RQ$_{1.1}$ Do classes affected by Sonar issues have a different level of *change-proneness* with respect to non-affected ones?

H$_{1.1.1}$ Classes affected by Sonar issues, *independently of their type and severity*, are more change-prone than non-affected ones.

H$_{1.1.0}$ The difference in change-proneness between classes affected by Sonar issues and non-affected classes is not statistically significant.

RQ$_{1.2}$ Do classes affected by Sonar issues have a different level of *fault-proneness* with respect to non-affected ones?

H$_{1.2.1}$ Classes affected by Sonar issues, *independently of their type and severity*, are more fault-prone than non-affected ones.

H$_{1.2.0}$ The difference in fault-proneness between classes affected by Sonar issues and non-affected classes is not statistically significant.

RQ$_2$ Do classes affected by Sonar issues *of different types* have a different level of change-proneness and fault-proneness with respect to non-affected ones?

In this RQ we aim to understand if the presence of Sonar issues of different types have a higher chance of being change-prone (RQ$_{2.1}$) or fault-prone (RQ$_{2.2}$). As an example, since Sonar issues classified as *Code Smells* are expected to increase the maintenance complexity and the maintenance effort, and Sonar issues classified as *Bugs* to increase fault likelihood, we expect a class containing *Code Smells* to be more change-prone while we expect classes affected by *Bugs* to be more fault-prone. Therefore, we formulate our sub RQs and their related hypotheses as follows:

RQ$_{2.1}$ Do classes affected by Sonar issues *of different types* have a different level of *change-proneness* with respect to non-affected ones?

H$_{2.1.1}$ Classes affected by Sonar issues, *of different types*, are more change-prone than non-affected ones.

H$_{2.1.0}$ The difference in change-proneness between classes affected by Sonar issues *of different types* and non-affected classes is not statistically significant.

RQ$_{2.2}$ Do classes affected by Sonar issues *of different types* have a different level of *fault-proneness* with respect to non-affected ones?

H$_{2.2.1}$ Classes affected by Sonar issues, *of different types*, are more fault-prone than non-affected ones.

H$_{2.2.0}$ The difference in fault-proneness between classes affected by Sonar issues *of different types* and non-affected classes is not statistically significant.

RQ$_3$ Do classes affected by Sonar issues of *different severity* have a different level of change-proneness and fault-proneness with respect to non-affected ones?

This RQ aims at understanding whether the severity level assigned to the different rules increases together with their actual change-proneness (RQ$_{3.1}$) or fault-proneness (RQ$_{3.2}$), independently of their type (*Bugs* or *Code Smells*). We hypothesized that classes affected by Sonar issues with higher levels of severity are more change-prone (H$_{3.1.1}$) or fault-prone (H$_{3.2.1}$) than those with a lower levels of severity. As an example, it might be possible that a highly severe *Code Smell* can turn into a fault, due to the high number of changes. The same could also apply for a piece of code that contains *Bugs* and may frequently be modified.

Therefore, we formulate our sub RQs and their related hypotheses as follows:

RQ$_{3.1}$ Do classes affected by Sonar issues of *different severity* have a different level of *change-proneness* with respect to non-affected ones?

H$_{3.1.1}$ Classes affected by Sonar issues of *different severity* are more change-prone than non-affected ones.

H$_{3.1.0}$ The difference in change-proneness between classes affected by Sonar issues of *different severity* and non-affected classes is not statistically significant.

RQ$_{3.2}$ Do classes affected by Sonar issues of *different severity* have a different level of *fault-proneness* with respect to non-affected ones?

H$_{3.2.1}$ Classes affected by Sonar issues, of *different severity*, are more fault-prone than non-affected ones.

H$_{3.2.0}$ The difference in fault-proneness between classes affected by Sonar issues of *different severity* and non-affected classes is not statistically significant.

RQ$_4$ Do classes affected by Sonar issues of *different types and severities* have a different level of change-proneness and fault-proneness with respect to non-affected ones?

This RQ aims at combining RQ$_2$ and RQ$_3$ to understand an eventual disagreement in the classification of Sonar issues and in their type and severity.

SonarQube classifies issues of type *Code Smell* as change-prone and issues of type *Bug* as fault-prone. Moreover, a higher level of severity implies a higher likelihood and impact of the problem underlined. As an example, an issue of type *Code Smell* with severity *Blocker*, will have a very high probability to have a very high impact on change-proneness. Similarly, an issue of type *Bug* with a higher severity will also have a high probability to lead to a very severe fault.

Therefore, we formulated two sub-RQs, to investigate the change-proneness ($RQ_{4.1}$) and the fault-proneness ($RQ_{4.2}$) of classes affected by SonarQube issues of different severity levels.

Based on the aforementioned assumptions, we expect the fault-proneness of classes affected by *Bugs* with higher severity to be more fault-prone that classes affected with lower levels of severity$H_{4.2.1}$). In addition, we expect that classes affected by *Code Smells* with higher severity to be more change-prone than classes affected with lower severity level items $H_{4.1.1}$).

Based on these assumptions, we formulated the following sub-research questions.

RQ$_{4.1}$ Do classes affected by Sonar issues of type *Code Smell* of *different severities* have a different level of *change-proneness* with respect to the severity level?

H$_{4.1.1}$ Classes affected by Sonar issues of type *Code Smells* with higher severity are more change-prone than classes affected by *Code Smells* with lower severity of non-affected classes.

H1$_{4.1.0}$ The difference in change-proneness between classes affected by *Code Smells* with different levels of severity is not statistically significant.

RQ$_{4.2}$ Do classes affected by Sonar issues of type *Bug* of *different severities* have a different level of *fault-proneness* with respect to non-affected ones?

H$_{4.2.1}$ Classes affected by Sonar issues of type *Bugs* with higher severity are more change-prone than classes affected by *Bugs* with lower severity of non-affected classes.

H1$_{4.2.0}$ The difference in change-proneness between classes affected by *Bugs* with different levels of severity is not statistically significant

5.2. Context

For this study, we adopted the projects included in the Technical Debt Dataset [34]. The data set contains 33 Java projects from the Apache Software Foundation (ASF) repository[8]. The projects in the data set were selected based on "criterion sampling" [62], that fulfill all of the following

[8]http://apache.org

criteria: developed in Java, older than three years, more than 500 commits and 100 classes, and usage of an issue tracking system with at least 100 issues reported. The projects were selected also maximizing their diversity and representativeness by considering a comparable number of projects with respect to project age, size, and domain.

Moreover, the 33 projects can be considered mature, due to the strict review and inclusion process required by the ASF. Moreover, the included projects regularly review their code and follow a strict quality process[9].

In Table 1, we report the list of the 33 projects we considered together with the number of analyzed commits, the project sizes (LOC) of the last analyzed commits, and the number of artifacts in the commits. More details on the data set can be found in [34].

5.3. Data Collection

As stated in the previous section, this study uses data from the Technical Debt Dataset [34]. The data set contains the analysis results from SonarQube for the projects we used in this study. The Sonar issue data in the data set was obtained by analyzing the commits of the projects using SonaQube's default rule set, i.e. "sonar way". We exported the relevant data from the data set as CSV files and used those in the actual analysis. All of the extracted data is available in the replication package [4].

In order to calculate the change-proneness of the classes, we needed the number of changes between two commits. We obtained this data from the Git repositories using Git's log command. We defined the change between two commits as the sum of the number of additions and deletions, as in the replicated study [10]. This data was not saved to a file as it was extracted from the project repositories during the analysis.

For calculating the fault-proneness of the classes, we needed to identify the fault-inducing and bug-fixing commits. These are also included in the Technical Debt Dataset and that data was exported from the dataset as well. In the dataset, the fault-inducing and fault-fixing commits are determined using the SZZ algorithm [31]. The steps of the algorithm are summarized as follows:

- **Step 1. Identification of bug-fixing commits**. As a part of this step, the algorithm matches commits with bug reports labeled as fixed, through regular expressions that allow identifying bug numbers and keywords in the commit messages.

[9]https://incubator.apache.org/policy/process.html

Table 1: Description of the selected projects

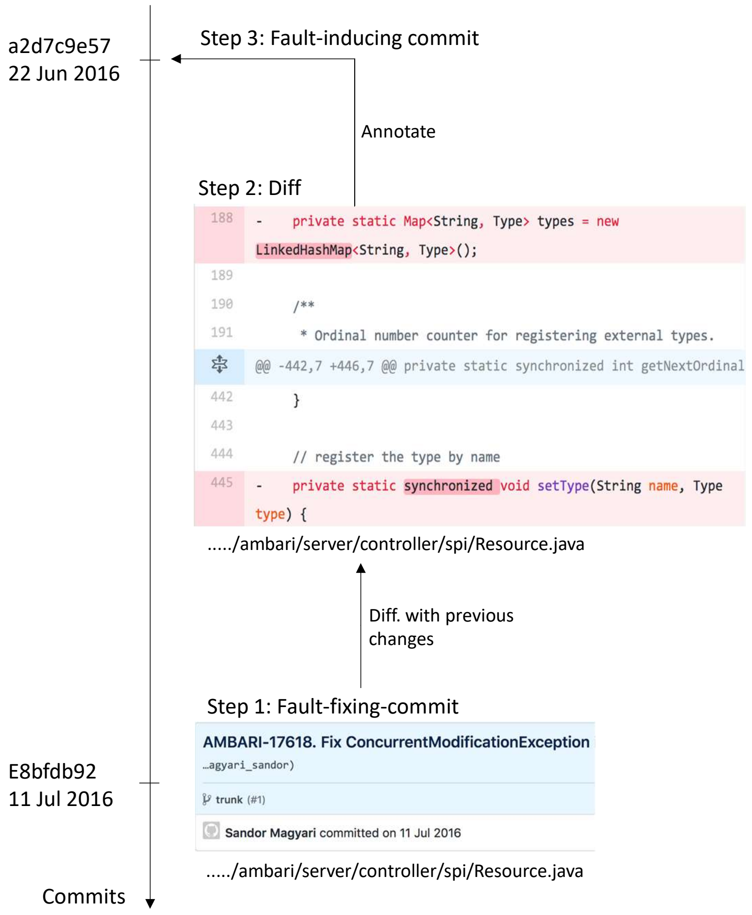
Project Name	**Analyzed Commits**		**Last Commit LOC**	**Last Commit Classes**	**# Faults**	**# Changes**
	#	**Timeframe**				
Accumulo	3	2011/10 - 2013/03	307,167	4,137	9,606	850,127
Ambari	8	2011/08 - 2015/08	774,181	3,047	7,110	677,251
Atlas	7	2014/11 - 2018/05	206,253	1,443	1,093	570,278
Aurora	16	2010/04 - 2018/03	103,395	1,028	19	485,132
Batik	3	2000/10 - 2002/04	141,990	1,969	54	365,951
Beam	3	2014/12 - 2016/06	135,199	2,421	51	616,983
Cocoon	7	2003/02 - 2006/08	398,984	3,120	227	2,546,947
Commons BCEL	32	2001/10 - 2018/02	43,803	522	129	589,220
Commons BeanUtils	33	2001/03 - 2018/06	35,769	332	1	448,335
Commons CLI	29	2002/06 - 2017/09	9,547	58	25	165,252
Commons Codec	30	2003/04 - 2018/02	21,932	147	111	125,920
Commons Collections	35	2001/04 - 2018/07	66,381	750	88	952,459
Commons Configuration	29	2003/12 - 2018/04	87,553	565	29	628,170
Commons Daemon	27	2003/09 - 2017/12	4,613	24	4	7,831
Commons DBCP	33	2001/04 - 2018/01	23,646	139	114	184,041
Commons DbUtils	26	2003/11 - 2018/02	8,441	108	17	40,708
Commons Digester	30	2001/05 - 2017/08	26,637	340	44	321,956
Commons Exec	21	2005/07 - 2017/11	4,815	56	40	21,020
Commons FileUpload	28	2002/03 - 2017/12	6,296	69	37	42,441
Commons IO	33	2002/01 - 2018/05	33,040	274	336	225,560
Commons Jelly	24	2002/02 - 2017/05	30,100	584	29	205,691
Commons JEXL	31	2002/04 - 2018/02	27,821	333	180	187,596
Commons JXPath	29	2001/08 - 2017/11	28,688	253	30	188,336
Commons Net	32	2002/04 - 2018/01	30,956	276	114	428,427
Commons OGNL	8	2011/05 - 2016/10	22,567	333	1	39,623
Commons Validator	30	2002/01 - 2018/04	19,958	161	60	123,923
Commons VFS	32	2002/07 - 2018/04	32,400	432	152	453,798
Felix	2	2005/07 - 2006/07	55,298	687	5,424	173,353
HttpComponents Client	25	2005/12 - 2018/04	74,396	779	15	853,118
HttpComponents Core	21	2005/02 - 2017/06	60,565	739	128	932,735
MINA SSHD	19	2008/12 - 2018/04	94,442	1,103	1,588	380,911
Santuario Java	33	2001/09 - 2018/01	124,782	839	99	602,433
ZooKeeper	7	2014/07 - 2018/01	72,223	835	385	35,846
Sum	**726**		**2,528,636**	**27,903**	**27,340**	**12,373,716**

- **Step 2. Identification of bug-introducing commit(s)**. As a part of this step, the algorithm first employs the diff functionality implemented in the control version systems to determine the lines that have been changed (to fix the bug) between the fixed commit version and its previous version.
  - Step 2.1. SZZ locates the commit that modified or deleted these lines the last time in previous change(s) applying annotate/blame functionalities. An example of Step 2.1 is presented in Figure 1. It shows the differences between the commit #e8bfdb and its predecessor (#300a7e) in the Resource.java file. In this case, in order to fix the bug, the data structure at line 188 was changed. Therefore, SZZ identifies the changes that introduced the bug AMBARI-17618 through the history of the source configuration management system (GitHub).
  - Step 2.2. SZZ labels the commit *#a2d7c9* as a potential bug-introducing commit.

The change-proneness and fault-proneness values presented in this paper have been normalized with the number of effective code lines in the Java file (see Section 5.4). To get this data, we used the grep command-line command. The number of lines for all Java files in each commit of the projects were saved in a pickle file. The file and the script for extracting the data are included in the replication package.

The data set contains analyzed data from 77,932 commits collected from 33 projects. However, we decided not to use all of those commits because of the definition of change-proneness and fault-proneness adopted in Palomba et al. [10] (see Section 5.4). The definitions compare the number of changes or faults between consequent commits so that if there is no difference between the classes, the value is zero. However, generally in a single commit, the vast majority of the classes are not changed. In addition, the frequency of commits can significantly vary between projects. Thus, if we would use all commits of a large and active project, we would detect that change-proneness is practically zero to almost all of the classes. This is the result of the size of the project and a large number of commits rather than actually not being prone to change. The same logic is applicable for the fault-proneness.

For this reason, we decided to fix the amount of time between commits that are used in the analysis. We used a commit from each of the projects once every 180 days (average time between releases in the analyzed projects). This makes sure a sufficient amount of time has passed in order to see the


a2d7c9e57
22 Jun 2016
Step 3: Fault-inducing commit
Annotate
Step 2: Diff
...../ambari/server/controller/spi/Resource.java
Diff. with previous changes
Step 1: Fault-fixing-commit
...../ambari/server/controller/spi/Resource.java
E8bfdb92
11 Jul 2016
Commits
trunk (#1)

```
-    private static Map<String, Type> types = new
LinkedHashMap<String, Type>();

     /**
      * Ordinal number counter for registering external types.
@@ -442,7 +446,7 @@ private static synchronized int getNextOrdinal
     }

     // register the type by name
-    private static synchronized void setType(String name, Type
type) {
```

AMBARI-17618. Fix ConcurrentModificationException
...agyari_sandor)

Sandor Magyari committed on 11 Jul 2016

Figure 1: SZZ Approach

changes in the classes. We would have preferred to use releases instead of fixing the time between commits, but we did not have the release information available in the data set. As shown in Table 1, out of the total 77,932 commits, we used 726 commits in our analysis (one every 180 days) and these commits had in total 200,893 Java class files.

5.4. Data Analysis

In order to answer our **RQs**, we investigated the differences between classes that are not affected by any Sonar issues (non-affected classes) and classes affected by at least one Sonar issue (affected classes). This paper compares the change-proneness and fault-proneness of the classes in these two groups.

We calculated the normalized class change-proneness and fault-proneness adopting the same approach used by Palomba et al. [10]. We considered the change-proneness of a class C_i in a commit s_k as:

$$change - proneness_{C_i,s_k} = \frac{\#Changes_{s_j \to s_k}(C_i)}{ELOC(C_i, s_k)} \tag{1}$$

where commit s_j precedes commit s_k; $\#Changes_{s_j \to s_k}(C_i)$ is the sum of additions and deletions made on C_i by developers during the evolution of the system between the commits s_j and the s_k; and $ELOC(C_i, s_k)$ is the number of effective lines of code in class C_i of commit s_k. We have defined an effective line of code as a non-empty line that does not contain only a curly bracket or start with "//", "/*", or "*".

Fault-proneness of class C_i in commit s_k is defined as follows:

$$fault - proneness_{C_i,s_k} = \frac{\#Fixes_{s_j \to s_k}(C_i)}{ELOC(C_i, s_k)} \tag{2}$$

where $\#Fixes_{s_j \to s_k}(C_i)$ is the number of commits between commits s_j and s_k that fixed a fault in the program and altered the class C_i in some way. $ELOC(C_i, s_k)$ is defined similarly as with the definition of change-proneness.

The results are presented using boxplots, which are a way of presenting the distribution of data by visualizing key values of the data. The plot consists of a box drawn from the 1^{st} to the 3^{rd} quartile and whiskers marking the minimum and maximum of the data. The line inside the box is the median. The minimum and maximum are drawn at 1.5*IQR (Inter-Quartile Range), and data points outside that range are not shown in the figure.

We also compared the distributions of the two groups using statistical tests. First, we determined whether the groups come from different distributions. This was done by means of the non-parametric Mann-Whitney test. The null hypothesis for the test is that when taking a random sample from two groups, the probability for the greater of the two samples to have been drawn from either of the groups is equal [63]. The null hypothesis was rejected and the distribution of the groups was considered statistically different if the p-value was smaller than 0.05.

As Mann-Whitney does not convey any information about the magnitude of the difference between the groups, we used also the Cliff's Delta effect size test. It is a non-parametric test meant for ordinal data. The results of the test were interpreted using guidelines provided by Grissom and Kim [64]. The effect size was considered negligible if $|d| < 0.100$, small if $0.100 \leq |d| < 0.330$, medium if $0.330 \leq |d| < 0.474$, and large if $|d| > 0.474$.

To answer **RQ1**, we compared the non-affected classes with all of the affected classes, while for **RQ2**, we grouped the affected classes based on the type of the identified Sonar issues and for **RQ3** by their level of severity. For each value of type and severity, we determined classes that were affected by at least one Sonar issue with that type/severity value and compared that group with the non-affected classes. Note that one class can have several Sonar issues and hence it can belong to several subgroups. For both **RQ2** and **RQ3** we used the same data, but in **RQ2** we did not care about the severity of the violated rule while on **RQ3** we did not care about the type.

Based on SonarQube's classification of issues, we expected that classes containing Sonar issues of the type *Code Smell* should be more change-prone, while classes containing *Bugs* should be more fault-prone. The analysis was done by grouping classes with a certain Sonar issue and calculating the fault-proneness and change-proneness of the classes in the group. This was done for each of the Sonar issues and the results were visualized using boxplots. As with **RQ2** and **RQ3**, each class can contain several Sonar issues and hence belong to several groups. Also, we did not inspect potential Sonar issue combinations.

To investigate **RQ4**, we grouped the data using both type and severity. We compared classes affected with Sonar issues of the same type and different severity levels. As in the previous **RQs** we visualized the distributions with boxplots and compared the distributions with Mann-Whitney and Cliff's Delta.

6. Results

RQ1. Do classes affected by Sonar issues have a different level of change-proneness and fault-proneness with respect to non-affected ones?

Out of all the issues detected by SonarQube, 173 were detected in the analyzed projects.

The analyzed commits contained 200,893 classes, of which 102,484 were affected by between 1 and 9,017 Sonar issues. As can be seen from Figure 2, most of the classes were either non-affected or affected only by one Sonar issue. In addition, the number of affected classes dropped almost logarithmically as the number of Sonar issues detected in the class grew. Thus, instead of presenting the results separately for each number of detected issues in the classes, we grouped the results. As the change was logarithmic, the sizes of the groups were determined using the values of function 2^n. This was done to make the groups more similar in terms of amount of data in the groups.

The distribution of the change-proneness of all of the classes in Figure 3 shows that the majority of the classes do not experience any changes and 75% of the classes experience less than 0.89 changes per line of code. The figure also suggests that the differences in the distributions are minor between the non-affected and affected classes.

In order to identify the significance of the perceived differences between the non-affected and the affected classes, we applied the Mann-Whitney and Cliff's Delta statistical tests. In terms of change-proneness, the p-value from the Mann-Whitney test was zero, which suggests that there is a statistically significant difference between the groups. The effect size was measured using Cliffs delta. We measured a d-value of -0.06, which indicates a small difference in the distributions (**RQ1.1**). Thus, we reject our null hypothesis $H_{1.1.0}$.

The fault-proneness of the classes is not visualized, as the number of faults in the projects is so small, and therefore the maximum value in the boxplot is also zero. Thus, all of the faults were considered as outliers. However, when the statistical tests were run with the complete data, the p-value from the Mann-Whitney test was zero. This suggests there is a statistically significant difference between the two groups. However, the effect size was negligible, with a d value of -0.005 (**RQ1.2**). Thus, we conclude that we do not have evidence for rejecting the null hypothesis $H_{1.2.0}$.

Moreover, we investigated the distributions of the change-proneness and fault-proneness of classes affected by different numbers of Sonar issues. We used the same groups as in Figure 2. The number of issues in a class does not seem to greatly impact the change-proneness (Figure 4). The only slightly different group is the group with 9-16 issues as its Q3 is slightly less than for the other affected groups.

The results from the statistical tests confirm that the number of Sonar issues in the class does not affect the change-proneness or fault-proneness of the class (Table 2). Considering change-proneness, the Mann-Whitney test suggested that the distribution would differ for all groups. However, the Cliff's Delta test indicated that the differences are negligible for all groups except the one with 17 or more items, for which the difference was small. Thus, differentiating the affected classes into smaller subgroups did not change the previously presented result (**RQ1.1**).

Once again, the fault-proneness is not visualized as the non-zero values were considered as outliers. In addition, while the statistical tests reveal that only the group with three or four Sonar issues was found to be similar to the non-affected group, all of the effect sizes were found negligible (**RQ1.2**).

Table 2: Results from the Mann-Whitney (MW) and Cliff's Delta tests when comparing the group of non-affected classes with groups of classes affected by different numbers of Sonar issues (RQ1)

#Sonar issues per class	**change-proneness**		**fault-proneness**	
	MW (p)	**Cliff (d)**	**MW(p)**	**Cliff (d)**
1	0.00	-0.048	0.00	0.009
2	0.00	-0.055	0.00	0.005
3-4	0.00	-0.061	0.82	-0.000
5-8	0.00	-0.075	0.00	-0.010
9-16	0.00	-0.063	0.00	-0.016
17→	0.00	-0.133	0.00	-0.036

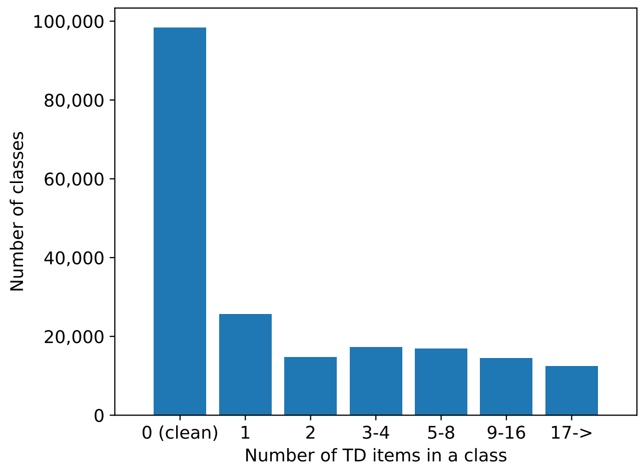


Figure 2: Number of classes with different numbers of Sonar issues. (RQ1)

However, the Cliff's Delta test indicated that the differences are negligible for all groups except the one with 17 or more items, for which the difference was small. In terms of fault-proneness, only the group with three or four Sonar issues was found to be similar to the non-affected group. However, all of the effect sizes were determined to be negligible.

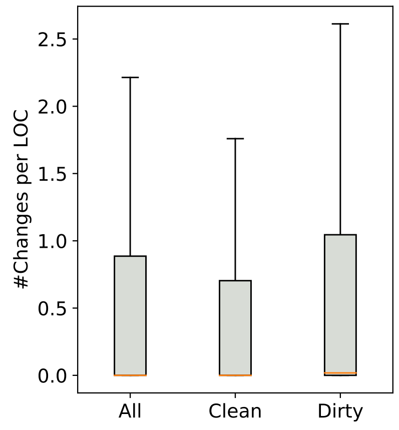


Figure 3: Change-proneness of classes affected by Sonar issues (RQ1)

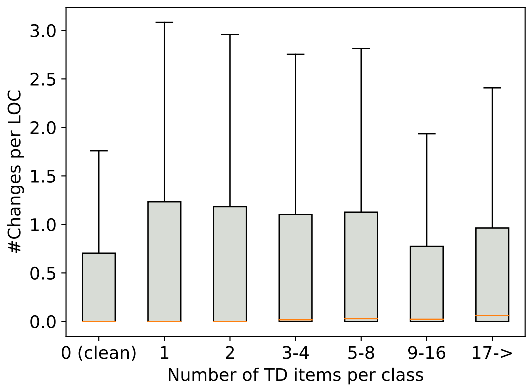
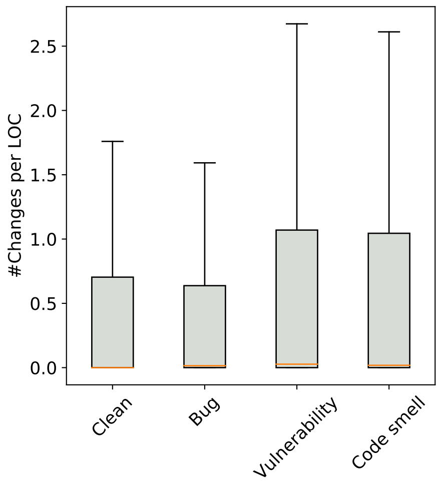
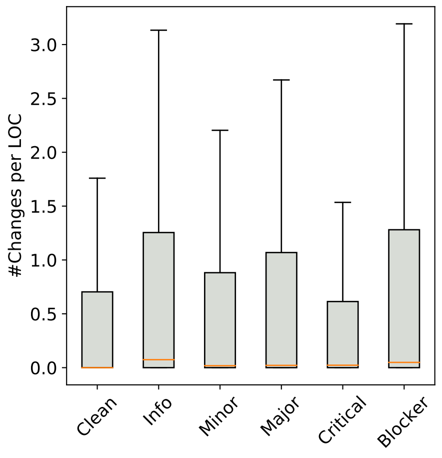


Figure 4: Change-proneness of classes affected by different numbers of Sonar issues (RQ1)



(a) Change-proneness (type)



(b) Change-proneness (severity)

Figure 5: Change-proneness of classes affected by Sonar issues considering type and severity (RQ2 and RQ3)

Summary of RQ1
The majority of the classes do not change and the rest of the classes have less than 2.5 changes per code line. Non-affected classes might be less change-prone than affected classes, but the difference between the groups is small. When inspecting for fault-proneness, the code generally does not contain any faults and there is no difference between the non-affected and affected classes. The number of Sonar issues in a class does not remarkably affect the change-proneness or fault-proneness.

RQ2. Do classes affected by Sonar issues of different types have a different level of change-proneness and fault-proneness with respect to non-affected ones?

The diffuseness of the detected 173 TD rules grouped by type is reported in Table 3. We collected data regarding the number of classes affected by each type and severity levels of Sonar issues (# **affected classes**). Moreover, we included the violated TD type recurrences (# **rules**) and how many times they are violated (# **introductions**).

The change-proneness of different types of affected classes is provided in Figure 5a. Fault-proneness is not visualized as the plot consists of only zeros. Looking at the change-proneness of the different types, the distributions are divided into two distinguishable groups. The most diffused types are *Vulnerability* and *Code Smell*, for which all of the key values are similar, with Q3 being approximately 1 and the maximum being 2.5. The less diffused groups are the *Bug* type and the non-affected classes, which are similar to each other in terms of Q3 and maximum. Moreover, the Mann-Whitney test suggested that regardless of the type, the distributions of the affected groups would differ from the distribution of the non-affected group. However, the measured effect size was negligible for all of the types (Table 4) (**RQ2.1**). Thus, we conclude that we accept the null hypothesis $H_{2.1.0}$

Regarding the evaluation of fault-proneness, the distribution for the number of faults per code line in a class consists only of zeros and outliers for all of the inspected groups. Thus it is not visualized in the paper. Moreover, there do not appear to be any significant differences between the affected and the non-affected groups. The Mann-Whitney test suggests that only the *Bug* type does not have a statistically significant difference in the distribution, with a p-value of 0.07. For the other types, the p-value was less than 0.01. However, the Cliff's Delta test suggests that all of the effect sizes are negligible as the $|d|$ values are smaller than 0.1 (**RQ2.2**). Thus, there is no evidence for rejecting the null hypothesis $H_{2.2.0}$.

Table 3: Diffuseness of detected Sonar issues (RQ2 and RQ3)

Type	**Severity**	**# Rules**	**# Affected Classes**	**# Introductions**
Bug	All	36	2,865	1,430
	Info	0	0	0
	Minor	0	0	0
	Major	8	1,324	377
	Critical	23	2,940	816
	Blocker	5	662	237
Code Smell	All	130	70,822	132,173
	Info	2	12,281	5,387
	Minor	32	70,426	44,723
	Major	80	78,676	73,894
	Critical	14	19,636	7,556
	Blocker	2	1,655	613
Vulnerability	All	7	3,556	2,241
	Info	0	0	0
	Minor	0	0	0
	Major	2	2,186	876
	Critical	5	3,490	1,365
	Blocker	0	0	0
Total		173	102,484	135,844

Table 4: Results from the Mann-Whitney (MW) and Cliff's Delta tests when comparing the group of non-affected classes with groups of classes affected by Sonar issues of different levels of severity and different types (RQ2 and RQ3)

Severity and Type		**Change-proneness**		**Fault-proneness**	
		MW (p)	**Cliff (d)**	**MW(p)**	**Cliff (d)**
Severity	Info	0.00	-0.144	0.00	-0.036
	Minor	0.00	-0.062	0.00	-0.009
	Major	0.00	-0.068	0.00	0.008
	Critical	0.00	-0.059	0.00	-0.018
	Blocker	0.00	-0.101	0.00	-0.066
Type	Bug	0.00	-0.054	0.07	-0.004
	Code Smell	0.00	-0.065	0.00	-0.005
	Vulnerability	0.00	-0.072	0.00	-0.022

Summary of RQ2
Considering the **type** of different Sonar issues, the types *Vulnerability* and *Code Smell* seem to be slightly more change-prone than the non-affected classes, but the differences are negligible. We did not find any significant differences regarding the fault-proneness of the classes.

RQ3. Do classes affected by Sonar issues of different severity have a different level of change-proneness and fault-proneness with respect to non-affected ones?

Table 3 reports the diffuseness of the detected 173 TD rules grouped by severity.

The change-proneness of the affected classes regarding different types is provided in Figure 5b. The most diffused levels are the least and the most severe levels *Info* and *Blocker*. Both of these groups have medians greater than zero, meaning most of the data does not consist of zeros. The median for *Info* is 0.07 and for *Blocker* it is 0.05, while their maximums are more than three changes per line of code and the Q3s are around 1.2. The least diffused level is *Critical*, while the levels *Major* and *Minor* are in the between.

The results from the Mann-Whitney and Cliff's Delta tests for the different severity levels are given in Table 4. The distribution of the change-proneness of all the groups was found to be different from that for the non-affected group. However, the measured effect size was negligible for the severity levels *Minor*, *Major*, and *Critical*, and small for the levels *Info* and *Blocker*. The results from the statistical tests confirmed the visual results from the boxplots, namely, that there are no significant differences between the non-affected classes and the different values of severity (**RQ3.1**). Therefore, we accept the null hypothesis $H_{3.1.0}$

Considering the fault-proneness of the different severity levels, the results are similar to the fault-proneness of the different type values. The Mann-Whitney test suggests that the distributions would differ for all levels, but when the effect size was measured, it was found to be negligible for every level (**RQ3.2**). Thus, there is no evidence for rejecting the null hypothesis $H_{3.2.0}$

Summary of RQ3
Regarding **severity**, the affected classes are not significantly more change-prone than the non-affected classes either. We did not find any significant differences regarding the fault-proneness of the classes.

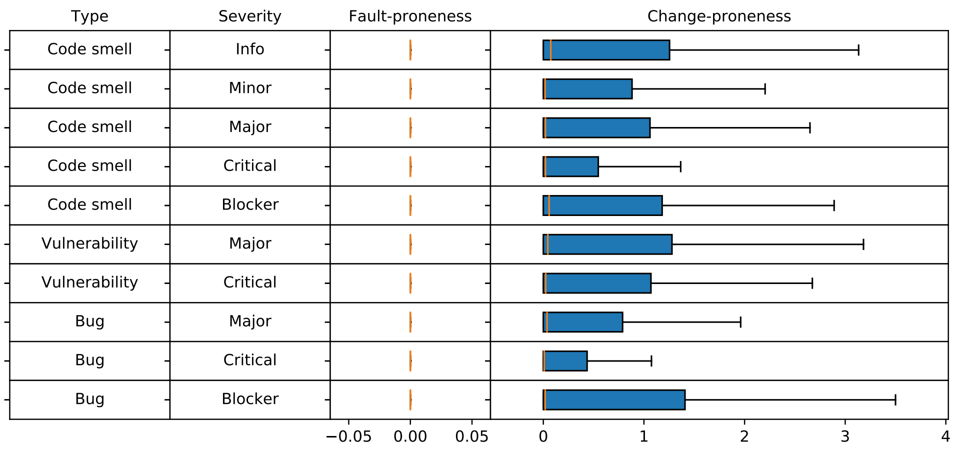


Figure 6: The change-proneness and fault-proneness of classes infected by different combinations types and severity levels (RQ4)

RQ4. Do classes affected by Sonar issues of different types and severities have a different level of change-proneness and fault-proneness with respect to non-affected ones?

Figure 6 presents the fault-proneness and change-proneness of the different type and severity combinations. Note that SonarQube does not have rules of type *Vulnerability* and severity *Info*, *Minor*, or *Blocker*; or of type *Bug* and severity *Info* or *Minor*.

Considering *Code Smells*, the change-proneness does not grow with the level of severity. Instead, surprisingly the most change-prone severity levels are *Info* and *Blocker* and the least change-prone is *Critical* (**RQ4.1**). We also inspected the change-proneness for the types *Vulnerability* and *Bug*. However, their change-proneness did not grow with the severity level.

Fault-proneness of the classes always is zero regardless of the type and severity level of the items affecting them. Thus, the fault-proneness of the type *Bug* was not affected by the severity level of the rules (**RQ4.2**)

Summary of RQ4
The change- or fault-proneness of the classes does not increase with the severity level regardless of the type of issues that are affecting them.

7. Discussion

In this Section, we discuss the results obtained according to the RQs and present possible practical implications from our research.

Answers to Research Questions. The analysis of the evolution of 33 Java projects showed that, in general, the number of Sonar issues in a class does not remarkably affect the change-proneness or fault-proneness. Non-affected classes (classes not affected by Sonar issues) might be less change prone than affected classes, but the difference between the groups is small. Moreover, when inspecting fault-proneness, the code generally does not contain any faults and there is no difference between the affected and non-affected classes.

Out of 266 different kind of issues detected by SonarQube, we retrieved 173 in the analyzed projects: 36 *Bugs*, 130 *Code Smells*, and 7 *Vulnerabilities.*

Taking into account Sonar issues classified as *Bug* (supposed to increase the fault-proneness) only one increases the fault-proneness. Unexpectedly, all the remaining *Bugs* resulted to slightly increase the change-proneness instead. As expected, all the 130 Sonar issues classified as *Code Smell* affect change-proneness, even if their impact on the change-proneness is very low. Moreover, also the seven Sonar issues classified as *Vulnerability* have a low effect on change-proneness.

However, the change-proneness and fault-proneness of the vast majority of Sonar issues (more than 70%) does not always increase together with the severity level assigned by SonarQube.

Implications. SonarQube recommends manual customization of their set of rules instead of using the out-of-the-box ruleset. However, as reported by [7], querying the SonarQube public instance APIs[10], we can see that more than 98% of the public projects use the default ruleset ("sonar way"), probably because developers have no experience with customizing nor understand which rules are more change-prone or fault-prone.

Our results are similar to Tollin et al. [23], even if in our case the effect of Sonar issues on change-proneness is very low. Tollin et al. found an increase in change-proneness in classes affected by Sonar issues in two industrial projects written in C# and Javascript. However, they adopted C# and Javascript rules, which are different from the Java rules. The difference in the results regarding change-proneness could either be due to the different projects (33 open-source Java projects) or to the different rules defined for Java. However, since Tollin et al.'s study was conducted on industrial projects where the code and the raw data are not accessible, it is not possible to compare the results more in-dept. Results also confirm the trend

[10]https://docs.sonarqube.org/display/DEV/API+Basics

obtained in our preliminary investigation on fault-proneness of SonarQube rules [29].

The main implication for practitioners is that they should carefully select the rules to consider when using SonarQube, especially if they plan to invest effort to reduce the change-proneness or fault-proneness. We recommend that practitioners should apply a similar approach as the one we adopted, performing a historical analysis of their project and classifying the actual change-proneness and fault-proneness of their code, instead of relying on their perception of what could be fault-prone or change-prone.

8. Threats to Validity

In this Section, we will introduce the threats to validity, following the structure suggested Runeson and Höst [65].

Construct Validity. This threat concerns the relationship between theory and observation. We adopted the measures detected by SonarQube, since our goal was to analyze the diffuseness of Sonar issues in software systems and to assess their impact on the change-proneness and fault-proneness of the code, considering also the type of Sonar issues and their severity.

Unfortunately, several projects in our data set do not tag the releases. Therefore, we evaluated the change-proneness and fault-proneness of classes as the number of changes and bug fixes a class was subject to in a period of six months. We are aware that using releases could have been more accurate for faults. However, as Palomba et al. [10] highlighted, the usage of releases has the threat that time between releases is different and the number of commits and changes are not directly comparable. Unfortunately, Git does not provide explicit tags for several projects in our data set. We relied on the SZZ algorithm [31] to classify fault-inducing commits. We are aware that SZZ provides a rough approximation of the commits inducing a fix because of Git line-based limitations of Git and because a fault can be fixed also modifying a different set of lines than the inducing ones. Moreover, we cannot exclude the misclassification of Jira issues (e.g., a new feature classified as bug). As for the data analysis, we normalized change-proneness and fault-proneness per class using effective LOC. As alternative, other measures such as complexity could have been used.

The selected projects did not use SonarQube during the analysis timeframe. As well as for the vast majority of works on Fowler's code smells, including the study we replicated [10], developers did not use the rules adopted in the study. Our results reflect exactly what developers would obtain using SonarQube out of the box in their project, without customizing the rule-set.

Some of the projects only recently adopted SonarQube (September - December 2019). To corroborate our findings, the projects that started to use SonarQube are using the default rule-set[11].

Internal Validity. This threat concerns the internal factors of the study that may have affected the results. We are aware that static analysis tools detect a non-negligible amount of false positives [66], and also the SonarQube's detection accuracy for some rules might not be perfect. However, our goal was to replicate the same conditions adopted by practitioners when using SonarQube, analyzing the change-proneness and fault-proneness of the Sonar issues detected by the default rule-set. Therefore, we did not modify or remove any possible false positives, to accurately reflect the results that developers can obtain running SonarQube on the same projects

Some issues detected by SonarQube were duplicated, reporting the issue violated in the same class and in the same position but with different resolution times. We are aware of this fact, but we did not remove such issues from the analysis since we wanted to report the results without modifying the output provided by SonarQube. We are aware that we cannot claim a direct cause-effect relationship between the presence of Sonar issues and the fault-proneness and change-proneness of classes, that can be influenced by other factors, as this type of investigation would require a controlled experiment. We are also aware that classes with different roles (e.g., classes controlling the business logic) can be more frequently modified than others.

External Validity. This threat concerns the generalizability of the results. We selected 33 projects from the Apache Software Foundation, which incubates only certain systems that follow specific and strict quality rules. We selected these projects to maximize diversity and representativeness by considering a comparable number of project with respect to age, size, and domain. The selected projects stem from a very large set of application domains, ranging from external libraries, frameworks, and web utilities to large computational infrastructures. The application domain was not an important criterion for the selection of the projects to be analyzed, but in any case, we tried to balance the selection and pick systems from as many contexts as possible. Choosing only one or a very small number of application domains, or project with similar age or size, could have been an indication of the non-generality of our study, as only prediction models from the selected application domain would have been chosen.

[11]SonarQube Projects on the Apache Software Foundation `https://sonarcloud.io/organizations/apache/projects`

Since we are considering only open-source projects, we cannot speculate on industrial projects.

Moreover, we only considered Java projects due to the limitation of the used tools (SonarQube provides a different set of Sonar issues for each language) and results of projects developed ion different languages would have not been directly comparable.

9. Conclusion

In this paper, we studied the impact of Sonar issues on change-proneness and fault-proneness, considering also the type and severity, based on 33 Java systems from the Apache Software Foundation. We analyzed nearly 726 commits containing 27K faults and 12 million changes. The projects were infected by 173 SonarQube issues violated more than 95K times in more than 200K classes analyzed.

Our results revealed that dirty classes might be more prone to change than classes not affected by Sonar issues. However, the difference between the clean and dirty groups was found to be at most small regardless of the **type** and **severity**. When considering the fault-proneness of the classes, no significant differences were found between the clean classes and the groups with dirty classes. As for SonarQube classification of Sonar issues, all the Sonar issues, including all the *Bugs*, *Code Smells* and *Vulnerabilities* have a statistically significant, but very small effect on change-proneness. Only one out of 36 Sonar issues classified as *Bug* (supposed to increase the fault-proneness) has a very limited effect on fault-proneness.

Our study shows that SonarQube could be useful and Sonar issues should be monitored by developers since all of them are related to maintainability aspects such as change-proneness. Despite our results show that the impact on change-proneness of Sonar issues is very low, monitoring projects with SonarQube would still help to write cleaner code and to slightly reduce change-proneness. As recommended by SonarQube, we would not recommend to invest time refactoring Sonar issues if the goal is to reduce change-proneness or fault-proneness, instead we would recommend not to write new code containing Sonar issues. The result of this work can be useful for practitioners and help them to understand how to prioritize Sonar issues they should refactor. It can also be used by researchers to bridge the missing gaps, and it supports companies and tool vendors in identifying TD as accurately as possible.

Regarding future work, we plan to further investigate the harmfulness of SonarQube issues, also comparing them with other types of technical debt, including architectural and documentation debt. We are planning to replicate this work, adopting different analysis techniques, including machine learning. Moreover, we also plan to conduct a case study with practitioners to understand the perceived harmfulness of Sonar issues in the code, so as to accurately identify the most relevant TD issues [67]. Finally we are planning to investigate automated approach to identify change and fault proneness of SonarQube issues, so as to enable companies to integrate such kind of approaches in their CI/CD pipelines [68], [69].

References

[1] W. Cunningham, The wycash portfolio management system, in: Object-Oriented Programming Systems, Languages, and Applications (Addendum), OOPSLA 92, Association for Computing Machinery, New York, NY, USA, 1992, p. 2930.

[2] W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, J. Syst. Softw. 80 (2007) 1120–1128.

[3] A. Martini, J. Bosch, M. Chaudron, Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study, Information and Software Technology 67 (2015) 237 – 253.

[4] T. Besker, A. Martini, R. E. Lokuge, K. Blincoe, J. Bosch, Embracing technical debt, from a startup company perspective, in: Int. Conf. on Software Maintenance and Evolution (ICSME), pp. 415–425.

[5] V. Lenarduzzi, A. Sillitti, D. Taibi, A survey on code analysis tools for software maintenance prediction, in: 6th International Conference in Software Engineering for Defence Applications, Springer International Publishing, 2020, pp. 165–175.

[6] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, A. Ampatzoglou, How do developers fix issues and pay back technical debt in the apache ecosystem?, in: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER 2018), pp. 153–163.

[7] N. Saarimäki, V. Lenarduzzi, D. Taibi, On the diffuseness of code technical debt in java projects of the apache ecosystem, in: Second International Conference on Technical Debt, TechDebt 19, IEEE Press, 2019, pp. 98–107.

[8] M. Fowler, K. Beck, Refactoring: Improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc. (1999).

[9] F. A. Fontana, V. Lenarduzzi, R. Roveda, D. Taibi, Are architectural smells independent from code smells? an empirical study, Journal of Systems and Software 154 (2019) 139 – 156.

[10] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, A. D. Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering 23 (2018) 1188–1221.

[11] G. Digkas, A. C. M. Lungu, P. Avgeriou, The evolution of technical debt in the apache ecosystem, Springer, 2017, pp. 51–66.

[12] A. Yamashita, L. Moonen, Do developers care about code smells? an exploratory survey, in: 20th Working Conference on Reverse Engineering (WCRE 2013), pp. 242–251.

[13] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: International Conference on Software Maintenance and Evolution (ICSME 2014), pp. 101–110.

[14] D.Taibi, A.Janes, V. Lenarduzzi, How developers perceive smells in source code: A replicated study, Information and Software Technology 92 (2017) 223 – 235.

[15] R. Arcoverde, A. Garcia, E. Figueiredo, Understanding the longevity of code smells: Preliminary results of an explanatory survey, in: Workshop on Refactoring Tools, WRT 2011, pp. 33–36.

[16] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: International Conference on the Quality of Information and Communications Technology 2010, pp. 106–115.

[17] A. Lozano, M. Wermelinger, B. Nuseibeh, Assessing the impact of bad smells using historical information, in: Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting, IWPSE 2007, pp. 31–34.

[18] D. Rapu, S. Ducasse, T. Girba, R. Marinescu, Using history information to improve design flaws detection, in: Eighth European Conference on Software Maintenance and Reengineering, CSMR 2004., pp. 223–232.

[19] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering 43 (2017) 1063–1088.

[20] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, F. A. Fontana, A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools, 1904.12538, arXiv (2020).

[21] F. Khomh, M. Di Penta, Y. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: Working Conference on Reverse Engineering 2009, pp. 75–84.

[22] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering 17 (2012) 243–275.

[23] I. Tollin, F. Arcelli Fontana, M. Zanoni, R. Roveda, Change prediction through coding rules violations, in: 21st International Conference on Evaluation and Assessment in Software Engineering, EASE 2017, pp. 61–64.

[24] M. Gatrell, S. Counsell, The effect of refactoring on change and fault-proneness in commercial c# software, Sci. Comput. Program. 102 (2015) 44 – 56.

[25] M. D'Ambros, A. Bacchelli, M. Lanza, On the impact of design flaws on software defects, in: 2010 10th International Conference on Quality Software, pp. 23–31.

[26] A. Saboury, P. Musavi, F. Khomh, G. Antoniol, An empirical study of code smells in javascript projects, in: International Conference on Software Analysis, Evolution and Reengineering (SANER 2017), pp. 294–305.

[27] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, S. Romano, On the accuracy of sonarqube technical debt remediation time, in: 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2019), pp. 317–324.

[28] M. T. Baldassarre, V. Lenarduzzi, S. Romano, N. Saarimaki, On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube, in: Information Software System.

[29] V. Lenarduzzi, F. Lomio, D. Taibi, H. Huttunen, Are sonarqube rules inducing bugs?, International Conference on Software Analysis, Evolution and Reengineering (SANER 2020). Preprint: arXiv:1907.00376 (2019).

[30] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, D. Binkley, Are test smells really harmful? an empirical study, Empirical Softw. Engg. 20 (2015) 1052–1094.

[31] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?, MSR '05, ACM, New York, NY, USA, 2005, pp. 1–5.

[32] J. C. Carver, Towards reporting guidelines for experimental replications: A proposal, in: RESER10.

[33] G. R. G. Rodriguez-Perez, J. M. Gonzlez-Barahona, Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm, Information and Software Technology 99 (2018) 164 – 176.

[34] V. Lenarduzzi, N. Saarimäki, D. Taibi, The technical debt dataset, in: 15th conference on PREdictive Models and data analycs In Software Engineering, PROMISE '19, pp. 2 – 11.

[35] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, H. C. Gall, Context is king: The developer perspective on the usage of static analysis tools, 25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 (2018) 38–49.

[36] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering (2019).

[37] W. J. Brown, R. C. Malveau, H. W. S. McCormick, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis, John Wiley and Sons, 1998.

[38] S. Vaucher, F. Khomh, N. Moha, Y. Gueheneuc, Tracking design smells: Lessons from a study of god classes, in: 2009 16th Working Conference on Reverse Engineering, pp. 145–154.

[39] S. Olbrich, D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: International Symposium on Empirical Software Engineering and Measurement, pp. 390–400.

[40] T. Amanatidis, A. Chatzigeorgiou, A. Ampatzoglou, The relation between technical debt and corrective maintenance in php web applications, Information and Software Technology 90 (2017).

[41] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, D. Poshyvanyk, Api change and fault proneness: A threat to the success of android apps, in: 9th Joint Meeting on Foundations of Software Engineering, p. 477487.

[42] T. Hall, M. Zhang, D. Bowes, Y. Sun, Some code smells have a significant but small effect on faults, ACM Trans. Softw. Eng. Methodol. 23 (2014).

[43] M. R. M. Deligiannis, I. Shepperd, I. Stamelos, An empirical investigation of an object-oriented design heuristic for maintainability, Journal of Systems and Software (1999).

[44] C. J. Kapser, M. W. Godfrey, "cloning considered harmful" considered harmful: Patterns of cloning in software, Empirical Softw. Engg. 13 (2008) 645–692.

[45] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, M. Temmerman, Does god class decomposition affect comprehensibility?, in: IASTED Conf. on Software Engineering.

[46] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, M. Shaw, Building empirical support for automated code smell detection, in: International Symposium on Empirical Software Engineering and Measurement, ESEM '10, ACM, New York, NY, USA, 2010, pp. 8:1–8:10.

[47] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, P. Avgeriou, The effect of gof design patterns on stability: A case study, IEEE Transactions on Software Engineering 41 (2015) 781–802.

[48] C. Carrillo, R. Capilla, Ripple effect to evaluate the impact of changes in architectural design decisions, in: European Conference on Software Architecture: Companion Proceedings, ECSA 18.

[49] S. M. Olbrich, D. S. Cruzes, D. I. K. Sjoberg, Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems, in: IEEE International Conference on Software Maintenance, ICSM '10, pp. 1–10.

[50] A. Lozano, M. Wermellinger, Assessing the effect of clones on changeability, in: IEEE International Conference on Software Maintenance, ICSM '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 227–236.

[51] D. I. K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, T. Dyba, Quantifying the effect of code smells on maintenance effort, IEEE Trans. Softw. Eng. 39 (2013) 1144–1156.

[52] A. R. Sharafat, L. Tahvildari, A probabilistic approach to predict changes in object-oriented software systems, in: European Conference on Software Maintenance and Reengineering (CSMR'07), pp. 27–38.

[53] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, Predicting the probability of change in object-oriented systems, IEEE Transactions on Software Engineering 31 (2005) 601–614.

[54] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, A method for assessing class change proneness, in: International Conference on Evaluation and Assessment in Software Engineering, p. 186195.

[55] E. M. Arvanitou, A. Ampatzoglou, K. Tzouvalidis, A. Chatzigeorgiou, P. Avgeriou, I. Deligiannis, Assessing change proneness at the architecture level: An empirical validation, in: Asia-Pacific Software Engineering Conference Workshops (APSECW), pp. 98–105.

[56] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, A. van Deursen, Code smells for model-view-controller architectures, Empirical Software Engineering 23 (2018) 2121–2157.

[57] M. Kessentini, A. Ouni, Detecting android smells using multi-objective genetic programming, pp. 122–132.

[58] K. Tashima, H. Aman, S. Amasaki, T. Yokogawa, M. Kawahara, Fault-prone java method analysis focusing on pair of local variables with confusing names, in: Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 154–158.

[59] E. Arisholm, L. C. Briand, Predicting fault-prone components in a java legacy system, in: International Symposium on Empirical Software Engineering, Association for Computing Machinery, 2006, p. 817.

[60] D. Falessi, B. Russo, K. Mullen, What if i had no smells?, International Symposium on Empirical Software Engineering and Measurement (ESEM) (2017) 78–84.

[61] V. R. Basili, G. Caldiera, H. D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering (1994).

[62] M. Patton, Qualitative Evaluation and Research Methods, Sage, Newbury Park, 2002.

[63] W. Conover, Practical nonparametric statistics, Wiley series in probability and statistics, Wiley, 3. ed edition, 1999.

[64] R. Grissom, J. J. Kim, Effect Sizes for Research: A Broad Practical Approach, 2005.

[65] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Softw. Engg. 14 (2009) 131–164.

[66] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why dont software developers use static analysis tools to find bugs?, in: International Conference on Software Engineering, ICSE 13, p. 672681.

[67] V. Lenarduzzi, A. Martini, D. Taibi, D. A. Tamburri, Towards surgically-precise technical debt estimation: Early results and research roadmap, in: International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE 2019, pp. 37–42.

[68] V. Lenarduzzi, A. C. Stan, D. Taibi, D. Tosi, G. Venters, A dynamical quality model to continuously monitor software maintenance, in: 11th European Conference on Information Systems Management (ECISM2017).

[69] A. Janes, V. Lenarduzzi, A. C. Stan, A continuous software quality monitoring approach for small and medium enterprises, in: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ICPE 17 Companion, p. 97100.