On the Diffuseness of Code Technical Debt in Java Projects of the Apache Ecosystem

Nyyti Saarimäki
Tampere University
Tampere, Finland
nyyti.saarimaki@tuni.fi

Valentina Lenarduzzi
Tampere University
Tampere, Finland
valentina.lenarduzzi@tuni.fi

Davide Taibi
Tampere University
Tampere, Finland
davide.taibi@tuni.fi

Abstract*—*Background. **Companies commonly invest major effort into removing, respectively not introducing, technical debt issues detected by static analysis tools such as SonarQube, Cast, or Coverity. These tools classify technical debt issues into categories according to severity, and developers commonly pay attention to not introducing issues with a high level of severity that could generate bugs or make software maintenance more difficult.**
Objective. **In this work, we aim to understand the diffuseness of Technical Debt (TD) issues and the speed with which developers remove them from the code if they introduced such an issue. The goal is to understand which type of TD is more diffused and how much attention is paid by the developers, as well as to investigate whether TD issues with a higher level of severity are resolved faster than those with a lower level of severity. We conducted a case study across 78K commits of 33 Java projects from the Apache Software Foundation Ecosystem to investigate the distribution of 1.4M TD items.**
Results. **TD items introduced into the code are mostly related to code smells (issues that can increase the maintenance effort). Moreover, developers commonly remove the most severe issues faster than less severe ones. However, the time needed to resolve issues increases when the level of severity increases (minor issues are removed faster that blocker ones).**
Conclusion. **One possible answer to the unexpected issue of resolution time might be that severity is not correctly defined by the tools. Another possible answer is that the rules at an intermediate severity level could be the ones that technically require more time to be removed. The classification of TD items, including their severity and type, require thorough investigation from a research point of view.**

***Index Terms*—Technical Debt issues, SonarQube, Violations**

I. Introduction

Companies commonly spend time to improve the quality of the software they develop, investing effort into refactoring activities aimed at removing technical issues believed to impact software qualities. Technical issues include any kind of information that can be derived from the source code and from the software process, such as usage of specific patterns, compliance with coding or documentation conventions, architectural issues, and many others.

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].

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 by means of static analysis tools, which is why several companies have started to adopt Code TD analysis tools such as SonarQube, Cast, and Coverity, investing a relevant amount of their budget into refactoring activities recommended by these tools. This is certainly a very encouraging sign, where a software engineering research topic receives balanced attention from both communities.

SonarQube is one of the most frequently used open source Code TD analysis tools [3]. It allows Code TD management by monitoring the evolution of TD and alerting developers if certain TD items 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 TD items are resolved (e.g., through refactoring) [4]. SonarQube has been adopted by more than 85K organizations [1], including nearly 15K public open-source projects [2]. SonarQube analyzes 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 creating a ”code issue”. Even if developers are not sure about the usefulness of the rules, they do pay attention to their categories and priorities and tend to remove violations related to rules with a high level of severity [5] in order to avoid the possible risk of faults [5], [6].

In this work, we aim to understand the diffuseness resolution time of TD items of Java projects in the Apache Software Foundation (ASF) Ecosystem. The goal is to understand which type of TD is more diffused and how much attention is paid by the ASF developers, as well as to investigate whether TD items with a higher level of severity are resolved faster than those with a lower level of severity.

Structure of the paper. Section II describes the basic concepts underlying this work and Section III some related work done by researchers in recent years. In Section IV, we describe the design of our case study, defining the research

[1]https://www.sonarqube.org
[2]https://sonarcloud.io/explore/projects

questions, metrics, and hypothesis, and describing the study context with the data collection and data analysis protocol. In Section V, we present the achieved results and discuss them in Section VI. Section VII identifies the threats to the validity of our study, and in Section VIII, we draw conclusions and give an outlook on possible future work.

II. Background

Technical Debt is one of the most recent research topic in Software Maintenance and Evolution [7]. In this Section, we introduce the concept of Code Debt and briefly describe SonarQube.

A. Code Debt

Li et al. [8] conducted a systematic mapping study to understand the concept of Technical Debt and create an overview of the current state of the art regarding the management of Technical Debt (TD). They proposed a classification of ten types of Technical Debt at different levels, derived from the selected studies (96). In their classification, they included: Requirement TD, Architectural TD, Design TD, Code TD, Test TD, Build TD, Documentation TD, Infrastructure TD, and Versioning TD. According to the definition of Li et al. [8], Code Debt is related to "*poorly written code*", avoiding all the "*best coding practices*" or "*coding rules*". Fowler [9] defined bad code smells as symptoms of poor design. Code violations can be considered coding rules.

B. SonarQube

SonarQube is one of the most common open-source static code analysis tools for measuring code debt. SonarQube is provided as a service from the sonarcloud.io platform or can be downloaded and executed on a private server.

SonarQube calculates several metrics such as 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. In case the analyzed source code violates a coding rule, or if a metric is outside a predefined threshold (also named "gate"), SonarQube generates a "TD issue". The time needed to remove these issues (remediation effort) is used to calculate the remediation cost and the technical debt. SonarQube includes Reliability, Maintainability, and Security rules. Moreover, SonarQube claims that zero false positives are expected from the Reliability and Maintainability rules, while there could be some false positives [3].

Reliability rules, also named "bugs", create TD 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 smells" adopted in SonarQube does not refer to the commonly known term code smells defined by Fowler et al. [9] but to a different set of rules.

SonarQube also classifies the rules into five *severity* levels [4]:

- *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 security 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, s̈witchs̈tatements 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 (Section IV-E).

III. Related Work

In this Section, we report the most relevant works on the investigation about the diffuseness of TD issues. To the best of our knowledge, the vast majority of the papers investigate the distribution and evolution of code smells and none of the papers investigates SonarQube violations.

Vaucher et al. [10] considered in their study God Class code smells, focusing on whether they affect software systems for long periods of time and making a comparison with whether the code smell is refactored.

Olbrich et al. [11] investigated the evolution of two code smells, God Class and Shotgun Surgery. They found that the distribution over time of these code smells is not constant; they increase during some periods and decrease in others, without any correlation with project size.

In contrast, Chatzigeorgiou and Manakos [12] investigated the evolution of several code smells and found that the number of instances of code smells increases constantly over time. This was also confirmed by Arcoverde et al. [13], who analyzed the longevity of code smells.

The longevity of code smells in the source code can lead to several issues [14]. It is interesting to understand the reason for this identified by Tufano et al. [14]: They found that the code is affected by code smells from the beginning of the development process, while some code smells are also introduced during refactoring activities. Moreover, Tufano et al. also report that close to 80% of the 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.

[3]SonarQube Rules:https://docs.sonarqube.org/display/SONAR/Rules Last Access: Dec.2018

[4]SonarQube Issues and Rules Severity:' https://docs.sonarqube.org/display/SONAR/Issues Last Access: Dec.2018

Palomba et al. [15] conducted a study on 395 versions of 30 different open-source Java applications, investigating the diffuseness of 13 code smells and their impact on two software qualities: change- and fault-proneness. They analyzed 17.350 instances of 13 code smells, which were identified by applying a metric-based approach. Out of the 13 code smells, only seven were highly diffused smells; their removal would result in great benefit to the software in terms of change proneness. In contrast, the benefit regarding fault proneness was very limited or non-existent. So programmers should keep an eye on these smells and do refactoring where needed in order to improve the overall maintainability of the code.

Digkas et al. [16] investigated the evolution of Technical Debt over five years at the granularity level of weekly snapshots. They considered as context sixty-six open-source software projects of the Apache ecosystem. Moreover, they characterized the lower-level constituent components of Technical Debt. The results showed a significant increase in terms of size, number of issues, and complexity metrics of the analyzed projects. However, they also discovered that normalized Technical Debt decreased as the project metrics evolved. Digkas et al. [4] also investigated how Technical Debt accumulates as a result of software maintenance activities. As context, they selected fifty-seven 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 that are fixed. The results showed that the largest percentage of Technical Debt repayment is created by a small subset of issue types.

Amanatidis et al. [17] investigated the accumulation of Technical Debt in PHP applications focusing on the relation between debt amount and interest to be paid during corrective maintenance activities. They analyzed ten open-source PHP projects from the point of view of corrective maintenance frequency and corrective maintenance effort related to interest amount and found a positive correlation between interest and the amount of accumulated Technical Debt.

IV. Case Study Design

We designed our empirical study as a case study. 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.

A. Goal and Research Questions

The goal of this study was to identify the diffuseness of Technical Debt issues in the source code, considering also the type of Technical Debt issues and their severity.

Based on the aforementionsd goal, we derived the following Research Questions (RQs):

RQ1: What is the diffuseness of Technical Debt issues in software systems?
This is a preliminary research question aimed at assessing to what extent software systems are affected by Technical Debt issues.

RQ2: What is the diffuseness of Technical Debt issues in software systems considering different types and levels of severity?
With this RQ, we considered different types and levels of severity for each TD item, in order to determine how the rules are grouped between different values of severity and type, and what the relative distribution of different levels of severity and different types is in the analyzed projects.

RQ3: What is the lifespan of Technical Debt issues?
With this RQ, we aim to investigate the time needed to resolve different TD issues, so as to understand which issues are resolved faster than others.

B. Context

We selected projects for this study based on a "criterion sampling" [18]. The selected projects had to fulfill all of the following criteria:

- Developed in Java
- Older than three years
- More than 1000 commits
- More than 100 classes
- Usage of issue tracking systems with at least 100 issues

Moreover, as recommended by Nagappan et al. [19], we also tried to maximize diversity and representativeness by considering a comparable number of projects with respect to project age, size, and domain.

Based on these criteria, we selected 33 Java projects from the Apache Software Foundation (ASF) repository [5]. This repository includes some of the most widely used software solutions. The available projects can be considered industrial and mature, due to the strict review and inclusion process required by the ASF. Moreover, the included projects have to keep on reviewing their code and follow a strict quality process[6].

In Table I, we report the list of the 33 projects we considered together with the number of analyzed commits, the project size (LOC) of the last analyzed commits, and the total number of TD issues.

C. Data Collection

All selected projects were cloned from their Git repositories. Each commit was analyzed for TD items using SonarQube. We used SonarQube's default rule set. We considered as TD items all the violations, called Squids (S), suggested to be refactored by SonarQube.

As variables we extracted: SonarQube violations (Squid) and Issue Remediation time.

We exported the data by means of SQ APIs in a csv file. The data is available in the replication package (Section IV-E).

[5]http://apache.org
[6]https://incubator.apache.org/policy/process.html

D. Data Analysis

For **RQ1**, we first determined the total number of TD items in each commit and then in each project. We also normalized the #TD issues in a commit by dividing their number by KLOC. Moreover, in order to see how diffused the items are within the commits, we inspected the percentage of classes with at least one TD item.

In addition, we examined the correlations between the number of TD items and the LOC, #classes, and #methods in a commit. We used the Spearman rank correlation coefficient ρ [20], which measures how well a monotonic function can be fitted between two groups of values measured from the same samples. This is a non-parametric method and the values of ρ range between -1 and 1, where 1 means perfect positive monotonic association, -1 means perfect negative monotonic association, and 0 means there is no monotonic association between the groups.

For interpreting the other values of ρ, we used the following guideline suggested by Cohen [21]: no correlation if $0 \leq \rho < 0.1$, small correlation if $0.1 \leq \rho < 0.3$, medium correlation if $0.3 \leq \rho < 0.5$, and large correlation if $0.5 \leq \rho \leq 1$. Corresponding limits apply for negative correlation coefficients.

The statistical significance of the correlations was inspected using p-values. The null-hypothesis H_0 was that there is no association between the sets. It was rejected if the p-value was smaller than 0.01; if it was higher, we concluded that there is an association between the groups.

RQ2 investigates the diffuseness of different TD item types and severity levels. The approach used in RQ1 was applied to both TD item type and severity. In addition, we determined how the rules are divided between different values of severity and type, and what the relative distribution of different levels of severity and different types is in the inspected projects.

RQ3 examines the lifespan of TD items. We defined the number of days it took to remove an introduced TD item by analyzing subsequent commits using SonarQube. The results are presented for both type and severity of the item. The results are visualized using boxplots. and are presented for both type and severity of the items.

E. Replicability

In order to allow our study to be replicated, we have published the complete raw data in the replication package [7].

TABLE I: Description of the selected projects

Project Name	Analyzed Commits		Last Commit LOC	#TD Items (Thousands)
	#	Timeframe		
Accumulo	2,641	2011/10 - 2013/09	340,670	985,373
Ambari	13,397	2011/08 - 2015/11	801,712	135,522
Atlas	2,336	2014/11 - 2018/06	204,418	9,389
Aurora	4,012	2010/04 - 2018/06	105,952	2,778
Batik	2,097	2000/10 - 2002/06	151,067	21,760
BCEL	1,324	2001/10 - 2018/05	43,850	5,387
Beam	2,865	2014/12 - 2016/07	140,489	15,415
BeanUtils	1,192	2001/03 - 2018/06	35,769	3,572
Cocoon	10,210	2003/02 - 2007/02	398,710	145,344
Codec	1,726	2003/04 - 2018/06	21,936	1,378
Collections	2,982	2001/04 - 2018/10	66,504	10,054
Commons CLI	896	2002/06 - 2018/02	9,579	9,145
Commons Configuration	2,895	2003/12 - 2018/06	87,712	5,685
Commons Daemon	980	2003/09 - 2018/05	4,616	251
Commons DBCP	1,861	2001/04 - 2018/06	26,447	3,169
Commons DbUtils	645	2003/11 - 2018/05	8,456	170
Commons Digester	2,145	2001/05 - 2018/05	30,965	3,498
Commons Exec	617	2005/07 - 2018/05	4,815	156
Commons FileUpload	922	2002/03 - 2018/05	6,328	272
Commons HttpClient	2,867	2005/12 - 2018/06	74,411	9,102
Commons IO	2,118	2002/01 - 2018/06	33,534	2,913
Commons Jelly	1,939	2002/02 - 2018/05	28,688	3,501
Commons JEXL	1,551	2002/04 - 2018/05	28,508	5,085
Commons JXPath	597	2001/08 - 2018/05	28,688	1,656
Commons Net	2,088	2002/04 - 2018/05	30,965	12,902
Commons OGNL	608	2011/05 - 2018/06	22,567	1,285
Commons Validator	1,339	2002/01 - 2018/05	19,966	934
Commons VFS	2,067	2002/07 - 2018/05	32,462	2,534
Felix	596	2005/07 - 2006/10	85,385	3,189
HttpCore	1,941	2005/02 - 2017/08	62,149	4,102
Santuario	2,697	2001/09 - 2018/06	125,329	11,844
SSHD	1,370	2008/12 - 2018/06	96,664	3,225
ZooKeeper	411	2014/07 - 2018/06	74,232	1,992
Sum	**77,932**		**5,194,399**	**1,422,599**

V. Results

In this Section, we introduce the results of the analysis.

A. RQ1: diffuseness of Technical Debt items

Out of 266 TD items monitored by SonarQube, 162 were detected in the analyzed projects. For reasons of space, we report only the 40 most frequent ones.

The complete list is available in the replication package (Section IV-E). The distribution of the number of TD item introductions is illustrated in Figure 1. In the figure, the TD item squids on the x-axis are rule identifiers from SonarQube, except for item db, which is a rule called common-java:DuplicatedBlocks.

Table II contains the results for both RQ1 and RQ3. Under the title "Diffuseness of TD items", we report the percentage of affected commits for the 40 most frequently introduced TD items. The diffuseness between commits differs greatly. For example, there are 19 TD items that affect 90% or more of the analyzed commits, while 28 can be found in 10% of the commits or less. The 40 most frequently introduced TD items are diffused to the majority of the commits, with the exception

[7] http://www.tut.fi/tase/raw_data/2019-TechDebtDiffuseness.zip

TABLE II: The diffuseness, correlations, and issue resolution times for the 40 most introduced TD items (RQ1 and RQ3)

SQ rule ID	RQ1						RQ3		
	Diffuseness of TD Items			Correlation between TD Items and Commit Size			Issue Resolution Time		
	%Affected Commits	Avg Instances	Max Instances	ρ with #Classes	ρ with #Methods	ρ with LOCs	AVG #Days	max #Days	stdev
S00112	98	1,180	29,821	**0.72**	**0.77**	**0.76**	178	6,211	362
S1128	83	689	30,170	*0.49*	**0.55**	**0.54**	114	5,936	235
S1130	94	979	30,691	**0.79**	**0.82**	**0.83**	205	6,101	354
S1166	99	850	23,279	**0.71**	**0.73**	**0.74**	232	6,267	472
S1192	99	1,006	18,689	**0.72**	**0.75**	**0.77**	263	6,211	533
S134	99	888	26,252	**0.76**	**0.82**	**0.81**	219	5,964	366
S1213	98	690	18,717	**0.75**	**0.78**	**0.75**	220	6,273	473
db	99	452	12,680	**0.77**	**0.84**	**0.80**	181	5,970	452
S106	94	552	19,265	*0.35*	*0.44*	*0.43*	223	5,936	455
S1541	99	539	17,476	**0.73**	**0.81**	**0.79**	261	6,211	396
S1133	90	390	13,670	0.29	*0.36*	*0.32*	215	5,938	391
S1132	92	593	14,733	**0.72**	**0.80**	**0.77**	291	5,834	494
S1124	84	535	16,292	**0.57**	**0.64**	**0.62**	302	6,015	453
S125	97	429	12,984	**0.63**	**0.70**	**0.69**	305	6,096	545
S1197	93	424	13,563	0.20	0.27	0.26	314	6,211	510
S1123	77	293	13,619	*0.37*	*0.45*	*0.41*	226	6,099	418
S1186	93	374	12,089	**0.66**	**0.75**	**0.67**	289	5,938	544
S00117	89	335	10,169	**0.61**	**0.65**	**0.69**	312	5,797	571
S1312	72	318	11,163	**0.68**	**0.72**	**0.70**	261	5,938	492
S1104	80	315	9,824	**0.67**	**0.74**	**0.69**	321	5,529	472
S00108	83	190	11,117	**0.57**	**0.62**	**0.61**	222	6,266	512
S00122	77	225	4,078	**0.61**	**0.62**	**0.61**	443	6,268	909
S00116	88	286	5,325	**0.58**	**0.62**	**0.63**	415	6,266	792
S1226	90	244	6,041	**0.59**	**0.68**	**0.64**	298	5,936	507
S1135	95	179	3,787	**0.80**	**0.80**	**0.81**	205	5,711	435
S1155	80	284	10,442	**0.78**	**0.82**	**0.79**	279	6,095	320
S1151	86	248	8,092	**0.65**	**0.70**	**0.70**	310	5,791	424
S00115	72	188	6,391	**0.69**	**0.75**	**0.77**	264	5,285	421
S1199	78	146	4,879	*0.48*	**0.54**	**0.53**	341	5,648	506
S1181	83	207	7,497	**0.57**	**0.61**	**0.59**	373	6,101	490
S2131	65	167	5,560	**0.60**	**0.65**	**0.65**	251	3,362	305
S1117	88	180	4,634	**0.78**	**0.85**	**0.80**	314	6,211	570
S00107	66	152	5,389	**0.85**	**0.86**	**0.86**	235	3,459	286
S00119	31	41	978	*0.48*	**0.51**	*0.48*	49	2,548	110
S1172	89	124	3,758	**0.73**	**0.80**	**0.78**	229	5,938	337
S1148	85	145	4,985	**0.66**	**0.71**	**0.72**	286	5,935	396
S1141	81	154	5,372	**0.66**	**0.74**	**0.74**	282	5,938	340
S00105	72	67	1,392	**0.50**	**0.57**	**0.56**	268	4,947	577
S1066	88	142	4,956	**0.71**	**0.77**	**0.75**	314	5,938	469
S1161	76	111	3,589	0.24	*0.32*	0.27	600	5,938	951

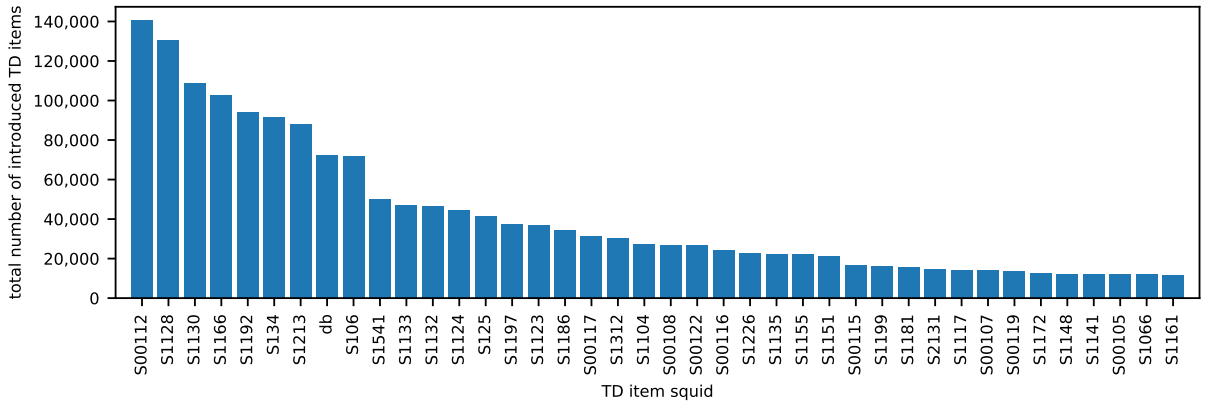
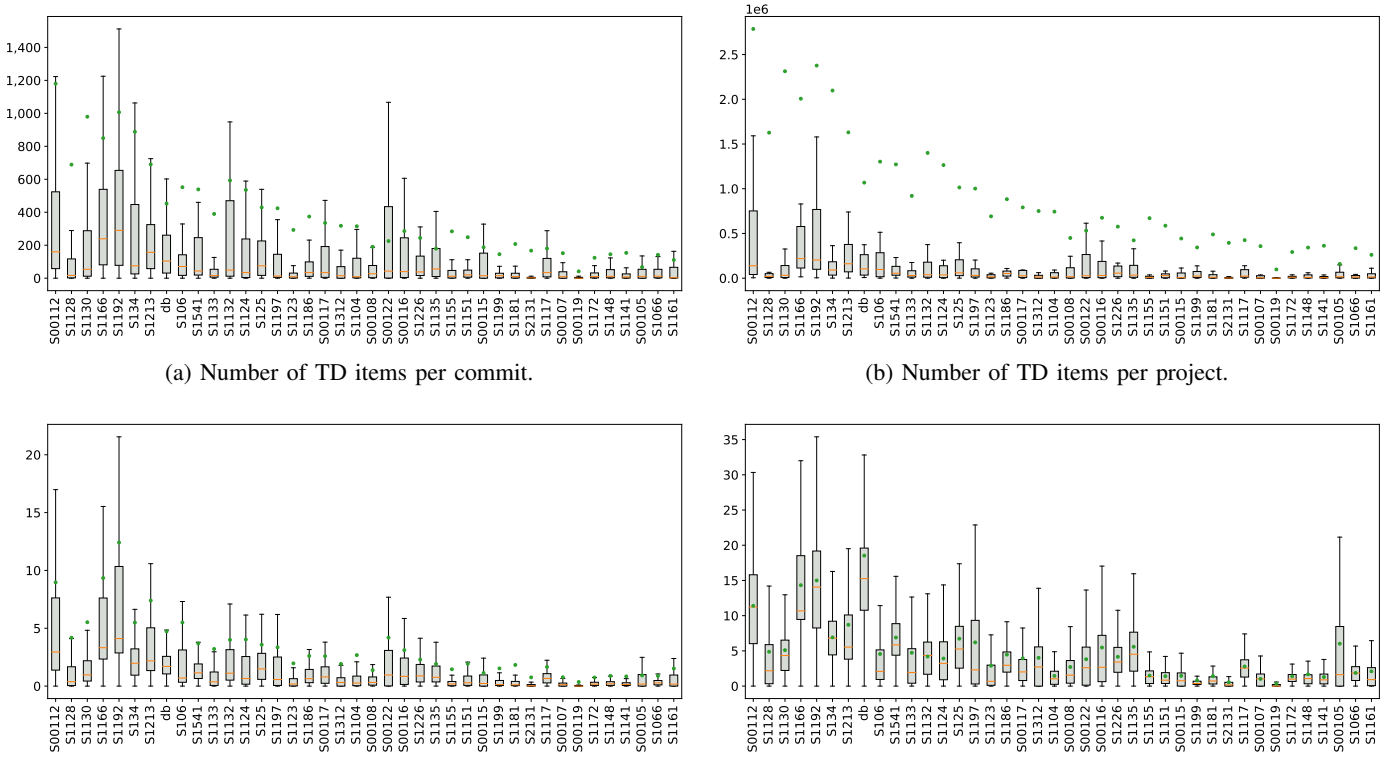


Fig. 1: Distribution of the 40 most frequently introduced TD items (RQ1)



(a) Number of TD items per commit.

(b) Number of TD items per project.

(c) Number of TD items per KLOC per commit.

(d) Percentage of affected classes per commit.

Fig. 2: Distributions of TD items (RQ1)

of S00119, which can be found only in 31% of the commits.

Figure 1 shows that the distribution varies significantly between items. Three items were introduced more than 100K times, while seven TD items were introduced between 50K and 100K times, 33 between 10K and 50K times, and 67 were introduced less than 1K times. Among the ones introduced less than 1K times, several were introduced only in a few projects and less than 10 times.

The diffuseness of the TD items is visualized using boxplots in Figure 2. Boxplots are a common way of displaying distribution by visualizing the minimum, first quartile (Q1), median, third quartile (Q3), and maximum of the data. The box marks the inter quartile range (IQR = Q3-Q1) and contains 50% of the data. The orange line inside the box is the median. We considered outliers all values outside the range ± 1.5*IQR. The black lines are called whiskers and represent the minimum and maximum, which in this case are the smallest/largest values not considered as outliers. The green dot marks the mean of the data. Figure 2 consists of four boxplots reporting #TD items per commit, #TD items per project, #TD items per KLOC per commit, and percentage of affected classes per commit. For reasons of readability, each boxplot contains only the 40 most frequently introduced TD items and outliers are not included in the boxplots. The complete list is available in the replication package (SectionIV-E).

As for the number of TD items per commit in Figure 2a, it can be observed that most of the boxplots are right-skewed. This means that larger values are distributed over a larger range while small values are located more closely to each other. Especially items S00112, S1166, S1192, S134, S1132, and S00122 have a wide distribution. However, not all of the most frequently introduced items have a wide range; for example, items S1128, S1133, and S1123 are each introduced in the systems over 35K times, but they have relatively low medians and narrow distributions. Many of the TD items have a median close to zero, this being true especially for the less frequently introduced items.

Taking into account the number of TD items per project in Figure 2b, we notice that means are generally larger than maximums. This is caused by large outliers. The boxplot suggests that some TD items are concentrated on certain projects, as the distribution is narrower when inspecting items per project versus items per commit; for example, items S1130 and S134. When we consider the TD items per thousands of lines of code (KLOC) in Figure 2c, we notice that the distribution of TD items is similar to what it was before normalization. However, there are fewer large outliers. The remarkable exceptions to the similarity of distributions are S1130 and S134, which have narrower distributions.

The percentage of affected classes per commit is presented in Figure 2d. In general, the percentage is low, as the highest median is 15% and the highest maximum 35%. The data has a limited number of outliers. There are also several items for which the distribution is not so remarkably right-skewed; for

example, S00112, S1192, and db.

In addition to inspecting the distributions of the TD items, we determined how well #TD items in a commit correlate with #classes, #methods, and LOC in that commit. The results are presented in Table II under the title ”Correlation between TD Items and Commit Size”. In the table, large correlations are marked in bold and medium correlations in italics. All p-values were under 0.01 except for two pairs: S2388 with #classes, and S2133 with LOCs. Neither of these two cases is one of the 40 most frequently introduced TD items.

The majority of the 40 most frequently introduced SQ rules have a strong positive correlation with all of the inspected system's size attributes. The exceptions are S106 and S1123, which have medium correlation, and items S1197, S1161, and S1149 with low or medium correlation. Both S1161 and S1149 have a narrow distribution with the median close to zero, whereas S1197 has a wider distribution when commits are inspected but a narrow distribution when projects are inspected. The correlations of the less frequently introduced rules were not as consistently high as those of the most frequently introduced rules.

B. RQ2: Diffuseness of TD items by type and severity

The distribution of SQ rules and introduced TD items is presented in Table III under the title ”Distribution”. When considering the type of items, the distribution is unbalanced. For example, 96% of all the items are code smells, even though 73% of the rules detected code smells by SonarQube. It is also worth noting that only 1% of the items were bugs, even though 23% of the rules were labeled as such.

The distribution for severity levels is more even. Most of the items are either minor or major items, as info, critical, and blocker constitute only 12% of the items. The distribution of rules and introduced items is similar.

The distribution of TD items in the analyzed commits considering type and severity of the items is presented in Table III under the title ”Percentage”. Regardless of whether considering severity or type, TD items are well distributed across the inspected commits. The only value affecting less than 93% of the commits is the severity level blocker, which impacts 86% of the commits.

Figure 3 shows the different distributions of all severity and type combinations by means of boxplots. Each row in the figure contains the data for a single type indicated on the left. In each plot, the distribution is drawn for all TD items of the type and separately for each severity level. Corresponding columns in the plots are visualizing the same aspect of the data and it is identified below the last plot. The type-severity combinations are drawn only when the used rule set has associated rules with that type and severity. As example, we do not draw the minor and info severity boxplots for bugs since we do not have rules in that level of severity.

When considering the number of TD items per project or commit, we notice that minor and major are the most diffused severity levels. On the other hand, the blocker level is hardly diffused at all, even though it affects 86% of the analyzed commits. In addition to blocker level, the info level is also diffused poorly. These two were the least frequently introduced levels, making up a total of up to 5% of all introduced items. The distribution is similar when projects are considered.

In general, code smells are much more diffused than bugs or vulnerabilities regardless of the visualized aspect. Comparing bugs and vulnerabilities reveals that bugs are a little less diffused than vulnerabilities. The normalized results reveal that there can be tens of code smells per KLOC but bugs and vulnerabilities are not found in every KLOC. The bug and vulnerability types have medians close to zero in all boxplots, even though they both affect at least 93% of the analyzed commits. The normalizing does not remarkably affect the distributions of the type code smell. For the bug type, the normalization reveals that the amount of critical bugs is relatively higher than the bugs of different severity level.

The results are similar when inspecting the percentage of affected classes per commit. The code smells are most diffused and in some cases there are major code smells in every class of the commit and the median is 31%. The bug and vulnerability types affect only a few percent of the classes regardless of the severity level.

The correlation coefficients between the value of severity/type and different aspects of the systems' size are presented in Table III under the title ”Correlation”. All severity and type values have large correlations with all inspected size measures, and all of the p-values are zero.

To see the ratio of different type and severity values in a project, the average distributions of level of severity and type are visualized in Figure 5. When considering the severity levels, the majority of projects have most of their TD items classified as minor or major, as expected based on their distribution. In addition, the number of critical and blocker items is less than 10% of all items. However, there are projects that have a higher rate of items with a high level of severity; for example, Collections and Commons IO have a rather high number of critical items, while BeanUtils has a high rate of both blocker and critical items.

When considering the type of the TD items, we see that almost all of them are code smells and that, for example, the project Codec contains code smells almost exclusively. The projects Commons Net and Zookeeper have a relatively high rate of vulnerabilities.

C. RQ3: What is the lifespan of TD issues

The number of days needed to fix TD items grouped by type and severity is presented in Table IV. The minimum number of days needed to remove an item is not presented, as it was 0 for all values. When considering the severity of an item, blockers have a notably higher median and average than the other levels. The critical, major, and minor levels are similar in terms of all reported measures. The info level differs from these in terms of the standard deviation, as for the info level this is 405, while the other values have a standard deviation over 470. As for the type of the items, bugs and vulnerabilities

TABLE III: Distribution of TD Items per type and severity (RQ2)

	Value	Distribution				Percentage			Correlation		
		# Available rules	# Detected rules	#Introduced items	% of introduced TD items	% affected commits	avg. #instances	max instances	ρ with #Classes	ρ with #Methods	ρ with LOCs
Severity	Info	3	2	69,516	4	98	569	15,699	**0.76**	**0.78**	**0.78**
	Minor	88	31	729,852	37	99	6,559	161,880	**0.78**	**0.84**	**0.83**
	Major	118	79	988,036	51	99	9,518	259,749	**0.77**	**0.86**	**0.82**
	Critical	42	42	144,394	7	99	1,321	33,979	**0.72**	**0.76**	**0.75**
	Blocker	15	8	18,101	1	86	229	8,214	**0.58**	**0.63**	**0.61**
Type	Code Smell	184	118	1,869,397	96	99	17,356	451,893	**0.79**	**0.86**	**0.84**
	Bug	64	37	22,690	1	96	205	6,388	**0.59**	**0.63**	**0.60**
	Vulnerability	18	7	57,812	3	93	636	17,837	**0.70**	**0.76**	**0.74**

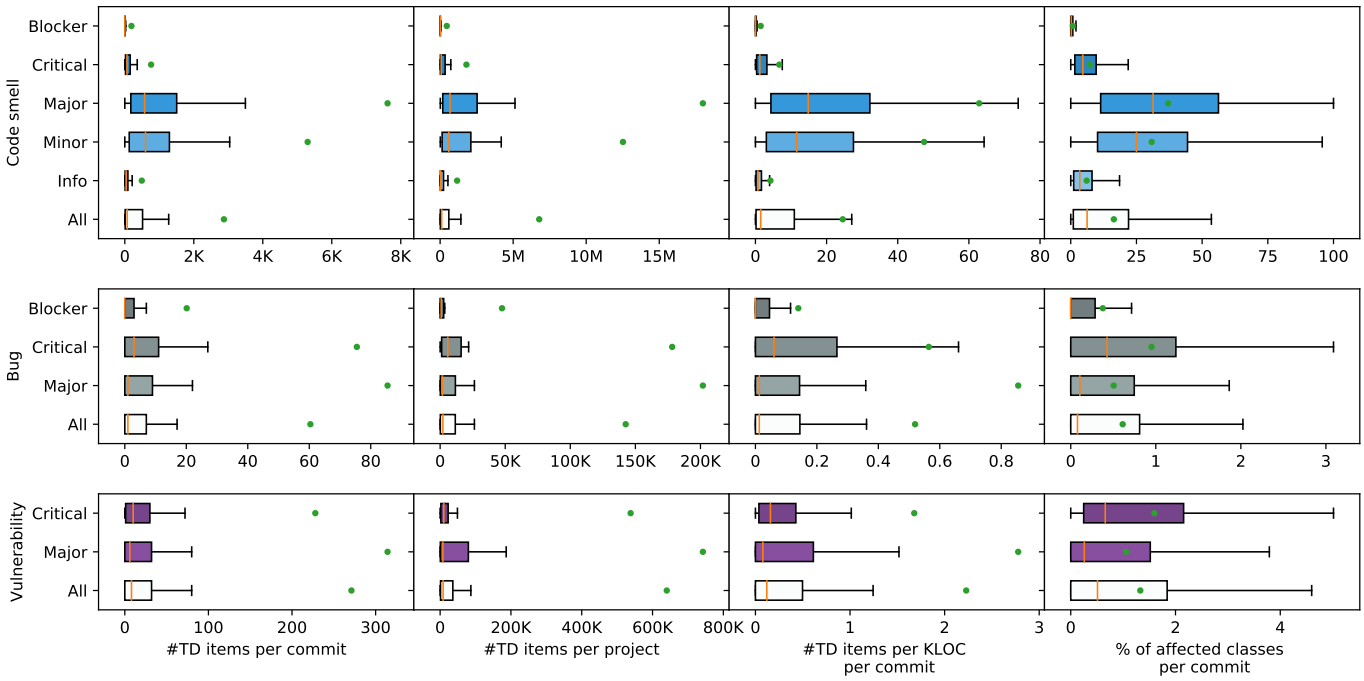


Fig. 3: The distributions TD items per type and severity (RQ2)

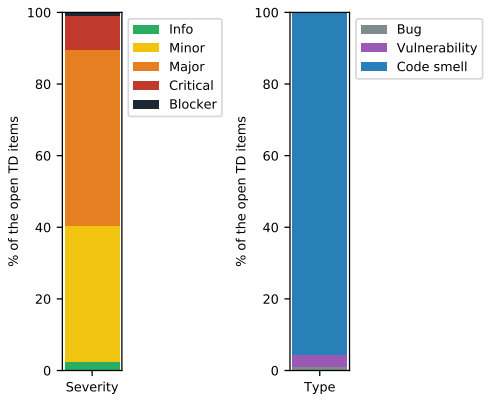


Fig. 4: Distribution of open TD items (RQ2)

are similar to each other. While code smells have a median of 60, bugs and vulnerabilities have a median of around 100.

The distribution of the open days is represented in Figure 6

TABLE IV: Number of days needed to resolve TD items (RQ3)

	Value	Median time	Avg. time	max	Stdev
Severity	Info	61	212	5938	405
	Minor	46	238	6273	482
	Major	72	259	6266	475
	Critical	75	257	6267	480
	Blocker	342	358	6101	486
Type	Code Smell	60	248	6273	476
	Bug	108	271	5775	425
	Vulnerability	102	310	5935	481

for all combinations of severity and type similarly to Figure 3, with the exception that the range on x-axis is shared for all plots. Major code smells have a significantly higher median than any other combination. It has median of 374, while the other combinations have medians around 100. All of the boxplots are right-skewed, even tough all of the means are below Q3 and have a value around 300 days.

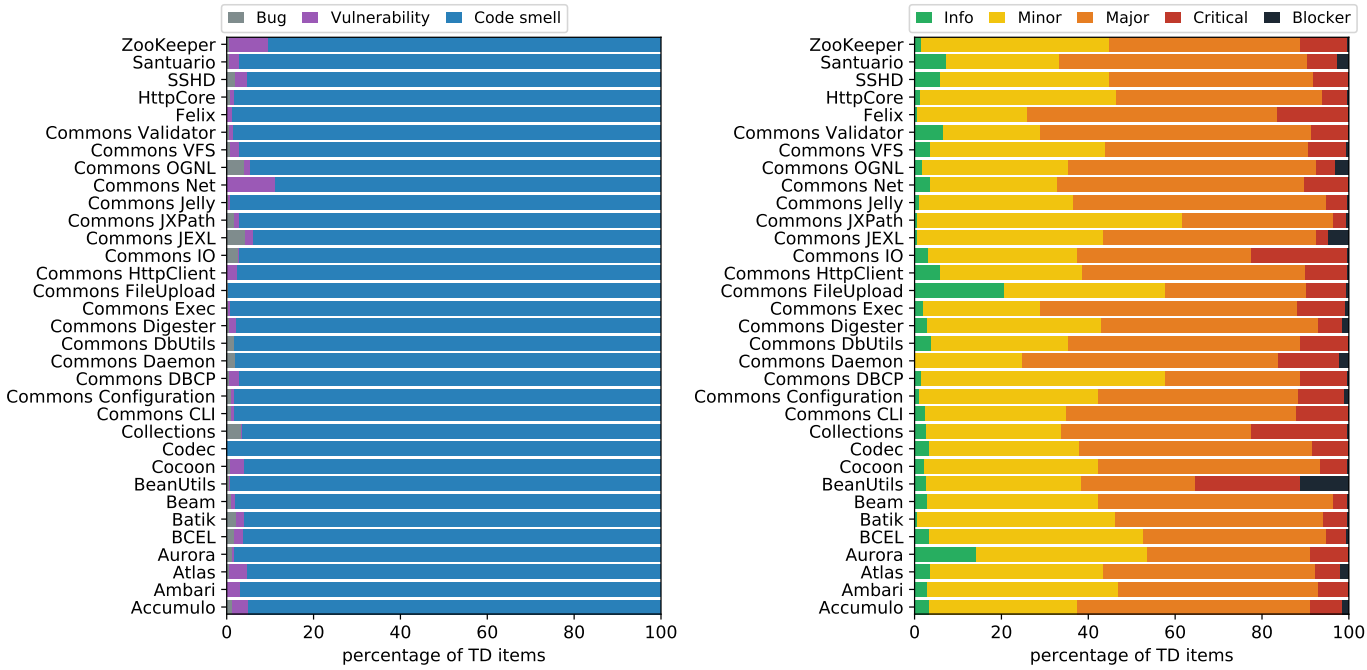


Fig. 5: The average contribution of the TD items in the analyzed projects considering the severity and type of the items (RQ2)

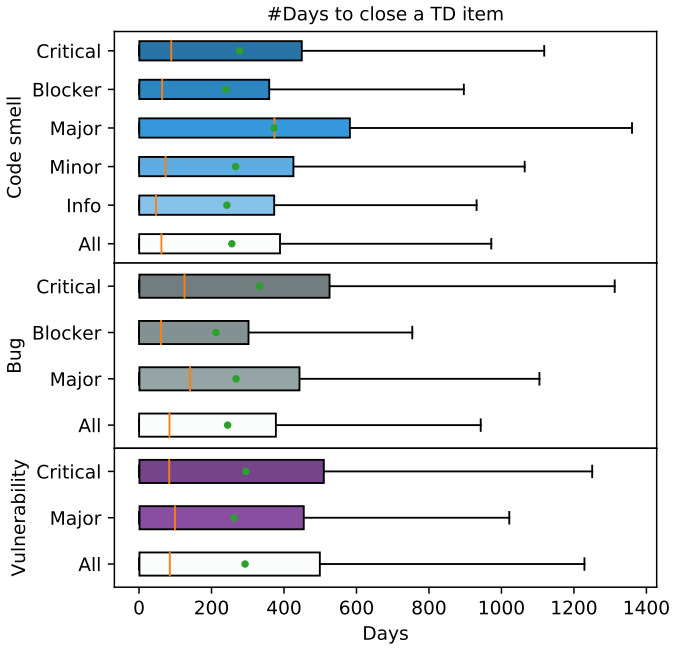


Fig. 6: The distribution of the number of days it takes to fix an introduced TD item based on item severity and type (RQ3)

VI. Discussion

There are rules which are never violated by the selected projects (36% of code smells, 42% of bugs, and 61% of vulnerabilities). Some rules are almost never found in the selected projects, while the most common rules are violated (generate a TD item) in almost every commit.

The most commonly introduced TD items correlate strongly with project size (LOC, #methods, and #classes). However, the normalization of the number of rules introduced per KLOC per commit did not have any significant impact on the distribution of the results.

As expected, since 73% of all detected rules are classified as code smells, 96% of the violated rules (TD items) are also code smells. As for the different severity levels of the rules, different projects violate rules with all severity levels. However, as might be expected, rules with the severity level blocker are slightly less diffused across the inspected commits than less severe ones.

Unexpectedly, the distribution of the severity level rules has a bell shape. We expected that the number of rules would decrease at higher levels of severity.

All levels of severity and all types have a strong correlation with the size-related measures LOC, #methods, and #classes.

The most unexpected result is related to the resolution time for the TD items. The time elapsed between the introduction and the removal of TD items classified as blockers is higher than that for other levels of severity. We can assume that this is because they are harder to fix than TD items with lower levels of severity, or they may be harder to detect, or developers might not even consider them as important as other issues that must be removed. When considering type and severity, the items with type code smell and severity major take the longest time to get resolved.

VII. Threats to Validity

Taking into account *Construct Validity*, we adopted the measures detected by SonarQube, since our goal was to validate the diffuseness of the rules produced by this tool. We are aware that the detection accuracy of some rules might not be perfect, but we tried to replicate the same conditions

adopted by practitioners when using the same tool. Threats to *Internal Validity* concern factors that could have influenced the results obtained. 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.

Threats to *External Validity* concern the generalization of the results obtained. We analyzed a relatively large number of projects and commits, and tried to select different projects with different characteristics. However, we are aware that other projects might present slightly different results.

VIII. Conclusion

In this paper, we studied the Code Technical Debt distribution of 33 Java systems from the Apache Software Foundation. We analyzed nearly 80K commits and mined more than 1.4M Technical Debt items reported by SonarQube. From this work we learned that a very small minority of problem types is responsible for the vast majority of estimated Technical Debt. As expected, the number of issues grows along with the size of the project.

As also highlighted by [16], some of the most frequently introduced TD issues are related to low-level coding issues, which could be decreased in several cases with good IDE support (e.g., duplicated strings) or with good IDE customization (e.g., two variables declared on the same line).

Although we expected most severe rules (blocker and critical) to be removed faster than less severe ones, the time elapsed between the introduction and the removal of the issues increased along with their severity.

Even though this study was conducted on a relatively large data set, we are aware that our results do not represent the whole Apache Software Foundation ecosystem. It would therefore be very valuable to replicate this study in larger contexts or in other open-source and industrial projects.

We are currently investigating the diffuseness and the impact of TD Issues in cloud-native applications [22], including SonarQube issues [23], patterns [24] and anti-patterns [25] [26] [24]. Future works include user studies on the investigation of the perceived harmfulness of TD Issues, with an approach similar to [6], understanding if the SonarQube TD issues can be used to predict external qualities such as the perceived reliability with approaches similar to these applied in [27] and [28].

References

[1] W. Cunningham, "The wycash portfolio management system," ser. OOPSLA '92, 1992.

[2] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, 2007.

[3] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *Software Engineering for Defence Applications — SEDA*, 2019.

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

[5] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," *Int. Conference on Software Analysis, Evolution and Reengineering (SANER)*, vol. 2018-March, pp. 38–49, 2018.

[6] D.Taibi, A.Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223 – 235, 2017.

[7] V. Lenarduzzi, A. Sillitti, and D. Taibi, "Analyzing forty years of software maintenance models," in *39th International Conference on Software Engineering Companion (ICSE-C '17)*, 2017, pp. 146–148.

[8] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, pp. 193–220.

[9] M. Fowler and K. Beck, "Refactoring: Improving the design of existing code," *Addison-Wesley Longman Publishing Co., Inc.*, 1999.

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

[11] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Int. Symp. on Empirical Software Engineering and Measurement*, 2009.

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

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

[14] P. F. Tufano, M., G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and A. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, Nov 2017.

[15] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, Jun 2018.

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

[17] T. Amanatidis, A. Chatzigeorgiou, and A. Ampatzoglou, "The relation between technical debt and corrective maintenance in php web applications," *Information and Software Technology*, vol. 90, 2017.

[18] M. Patton, *Qualitative Evaluation and Research Methods*. Newbury Park: Sage, 2002.

[19] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," ser. ESEC/FSE 2013, 2013, pp. 466–476.

[20] Student, "An Experimental Determination of the Probable Error of Dr Spearman's Correlation Coefficients," *Biometrika*, vol. 13, no. 2/3, p. 263, jul 1921.

[21] J. Cohen, "Statistical power analysis for the behavioral sciences," *Lawrence Earlbaum Associates*, vol. 2, 1988.

[22] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[23] N. Saarimäki, F. Lomio, V. Lenarduzzi, and D. Taibi, "Does migrate a monolithic system to microservices decreases the technical debt?" *CoRR*, vol. abs/1902.06282, 2019. [Online]. Available: http://arxiv.org/abs/1902.06282

[24] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study," *8th International Conference on Cloud Computing and Services Science (CLOSER2018)*, 2018.

[25] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[26] U. Azadi, F. Arcelli Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," *International Conference on Technical Debt (TechDebt 2019)*, 2019.

[27] L. Lavazza, S. Morasca, D. Taibi, and D. Tosi, "Predicting oss trustworthiness on the basis of elementary code assessment," in *Int. Symp. on Empirical software engineering and measurement - ESEM*, 2010.

[28] L. Lavazza, S. Morasca, D. Taibi, and D. Tosi, "An empirical investigation of perceived reliability of open source java programs," in *ACM Symposium on Applied Computing, SAC 2012*, 2012, pp. 1109–1114.