How long do Junior Developers take to Remove Technical Debt Items?

Valentina Lenarduzzi,[1] Vladimir Mandić,[2] Andrej Katin,[2] Davide Taibi[3]
[1]LUT University, Finland — [2]University of Novi Sad, Serbia — [3]Tampere University, Finland
valentina.lenarduzzi@lut.fi;vladman@uns.ac.rs;katin@uns.ac.rs;davide.taibi@tuni.fi

ABSTRACT

Background. Software engineering is one of the engineering fields with the highest inflow of junior engineers. Tools that utilize source code analysis to provide feedback on internal software quality, i.e. Technical Debt (TD), are valuable to junior developers who can learn and improve their coding skills with minimal consultations with senior colleagues. *Objective.* We aim at understating which SonarQube TD items junior developers prioritize during the refactoring and how long they take to refactor them. *Method.* We designed a case study with replicated design and we conducted it with 185 junior developers in two countries, that developed 23 projects with different programming languages and architectures. *Results.* Junior developers focus homogeneously on different types of TD items. Moreover, they can refactor items in a fraction of the estimated time, never spending more than 50% of the time estimated by SonarQube. *Conclusion.* Junior Developers appreciate the usage of SonarQube and considered as a useful tool. Companies might ask junior developers to quickly clean their code.

KEYWORDS

Technical Debt, SonarQube, Remediation Time, Junior Developers, Empirical Study

ACM Reference Format:
Valentina Lenarduzzi,[1] Vladimir Mandić,[2] Andrej Katin,[2] Davide Taibi[3]. 2020. How long do Junior Developers take to Remove Technical Debt Items?. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '20), October 8–9, 2020, Bari, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3382494.3422169

1 INTRODUCTION

Software engineering is one of the engineering fields with the highest inflow of junior engineers[1]. The disproportion of junior and senior developers is increasing fast, and it puts a significant stress on the mentoring and tutoring process. Under such conditions, tools that provide feedback on internal software quality through source code analysis—*automated static analysis tools* (ASAT)—are valuable to junior developers who can learn and improve their coding skills with minimal consultations with senior colleagues.

The popularity of such tools, to measure software quality detecting potential issues in the code, is rapidly increasing [17, 18]. Among the existing static analysis tools, SonarQube has been adopted by more than 100K organizations[2] including nearly more than 15K public open-source projects.[3]

One of the aspects of software quality is commonly depicted with technical debit (TD) metaphor [3]. TD contextualizes the problem of outstanding software development tasks (e.g. tests planned but not executed, pending code refactoring, etc.) as a kind of debt that brings a short-term benefit to the project that may have to be paid later in terms of increased effort or rework (e.g. a poorly designed class tends to be more difficult and costly to maintain than if it had been implemented using good object-oriented practices) [3]. Some empirical investigations indicate that junior developers are less familiar with the concept of TD than their more experienced colleagues [19].

SonarQube was one of the first ASAT tools to provide estimates of the accumulated TD in source code, i.e. needed effort to remediate issues that according to SonarQube's rules represent violations of good practices.

In this work, we aim to understand how long junior developers take to fix TD items. For this purpose, we designed and conducted a multiple case study with replicated design [29] involving 185 fourth-year graduate and master students with programming experience comparable to junior developers.

The contributions of this paper can be summarized as follows:

- Analysis of the diffuseness of TD items introduced by junior developers
- Identification of the type of TD items commonly refactored by junior developers
- Analysis of the time spent for removing TD Items
- Comparison of the actual and estimated TD Items refactoring time

The remainder of this paper is structured as follows: Section 2 describes the empirical study design. Section 3 presents and discusses results. Section 4 identifies the threats to validity of this work and Section 5 reports on related work. At the end, Section 6 draws the conclusion and highlights the future works.

2 THE CASE STUDY

In this section, we describe the empirical study designed according to the guidelines proposed by Runeson and Höst [22].

2.1 Goal and Research Questions

We formalized the goal of this study as follows:

[1]https://evansdata.com/

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

ESEM '20, October 8–9, 2020, Bari, Italy

ACM ISBN 978-1-4503-7580-1/20/10…$15.00
https://doi.org/10.1145/3382494.3422169

Analyze TD items
for the purpose of evaluating
with respect to their remediation time
from the point of view of junior developers
in the context of software development process

Based on the above-mentioned goal, we formulated the following Research Questions (RQ_s):

RQ$_1$ What TD items are introduced by junior developers?
RQ$_2$ Which TD items are commonly refactored by junior developers?
RQ$_3$ How long do junior developers spend to refactor TD items?
RQ$_4$ What is the remediation time accuracy?

In **RQ**$_1$ we aim at assessing what TD items junior developers are more prone to introduce in software systems during the development process. If TD items introduced are poorly diffused, the study might be not worthwhile. We focus on which TD items are mostly refactored for first, considering TD types defined by SonarQube (Bugs, Code Smells, and Security Vulnerabilities) and the severity level assigned to each TD item (**RQ**$_2$).

Furthermore, we aim at investigating how long junior developer spends time to refactor the TD items identified thru **RQ**$_3$. Moreover, we aim to measure the accuracy of the remediation time spent to refactor each TD item (**RQ**$_4$).

We hypothesize that junior developers fix TD items with less TD estimated and with a lower severity level assigned by the tools (**H**$_1$ - **RQ**$_3$). Moreover, we hypothesize that junior developers are more accurate fixing TD items with less debt estimated and with a lower severity level assigned by the tools (**H**$_1$ - **RQ**$_4$).

2.2 Study Design

In order to achieve our goal and to answer to our **RQ**$_s$, we designed a multiple case study with a replication design [29] executing one replication Tampere University (Replication 1) and executing another replication in University of Novi Sad (Replication 2).

In each Replication, we investigated the TD introduced by junior developers together with its related refactoring and remediation time.

Each replication has to create teams composed to 4-8 members that will develop a software project for three months. After 1.5 months, the projects will be analyzed with SonarQube to detect TD items. Developers are then required to refactor the code to fix TD items detected by SonarQube. The motivation to select Sonarqube and more details on its levels are available in the online appendix included in the replication package[4]. At the end of the project, we analyze the diffuseness of TD items introduced by developers (RQ_1), the refactoring made by developers (RQ_2-RQ_3) and its accuracy (RQ_4).

In order to allow our study to be replicated, we have published the complete raw data in the replication package[4].

2.3 Case and Subject Selection

Case Selection. We selected participants for the two replications from two University courses. Participants need to be master or fourth-year students with experience in software development. More than half of the participants need to have at least two years of full-time industrial experience as software developer. The course needs to develop a software project in the duration of three months and enable to develop the same project in teams. The courses need to use the same version of SonarQube, and collect data for this study using the same protocol. Courses are free to select the project topic independently.

Based on these requirements we identified two master courses: one master course at the Tampere University (Finalnd) and one master course at the University of Novi Sad (Serbia). The reason for selecting courses in different countries and with different instructors was to increase the generality of the results. Indeed, results will be more generalizable if cross-case conclusions are comparable.

Projects and Participant Selection. In this Section, we describe the projects and participants selection for both replications. The project assignment was mandatory for the course. The first members of the teams were picked by instructors, afterward each picked member was instructed to pick one next member from remaining students until the groups are formed. As a part of QA activities, students were instructed to use SonarQube (community Version 7.4 (build 18908) - more details about the adopted tool are reported in the online appendix included in the replication package[4]). SonarQube server was set-up and administered by course instructors.

Replication 1. The research was done in the context of a software engineering university course with 133 second-year master students at Tampere University.

The project assignments included the development of a tool to analyze all the commits of any Git-based projects with the three static analysis tools and export the results in a csv file. Performance and efficiency were considered the most important non-functional requirements. Students were organized in teams composed of 4-5 persons. Each team developed the same project.

The 133 students were organized in six groups of sizes from 4 to 6 members, for a total of 26 groups. For completing the assignments students were given two months (from the beginning of February 2019 until the end of April 2019), during this period they had regular weekly meetings with teaching staff.

Replication 2. The research was done in the context of a software engineering course with 52 fourth-year bachelor students at the University of Novi Sad.

The project assignments included a high-level specification of an e-commerce system and instructions for the development process. Students were expected to further refine given requirements specification, and to design and implement a solution using concepts of domain-driven design [7] and microservices [20]. Students were organized in teams, each team was responsible for a set of microservices—a system module. Students were given training on how to design microservice-based systems and how to develop microservices in .NET technology.

The 52 students were organized in 7 teams of sizes from 6 to 8 members. For completing the assignments students were given 3 months (from December 2018 until March 2019), during this period they had regular bi-weekly meetings with teaching staff.

[4] https://figshare.com/s/f5d0a0a3ee0a6b23468e

2.4 Data Collection

Student teams that successfully completed the development phase of the project assignment were given an excel file with the lists of open issues that need to be fixed, plus two columns for collecting needed time (effort) to fix an issue alongside with status of the fix. Students were asked to resolve as many issues as they can, and report back spent effort and status of issue fixes. Upon receiving reports, SonarQube analysis is performed on committed code by instructors to validate reported issue fixes.

2.5 Data Analysis

We applied three accuracy evaluation criteria: *RE* (Relative Error), *MMRE* (Mean Magnitude Relative Error), *MdMRE* (Median Magnitude Relative Error). The usage of different indicators allows us to obtain a more complete picture of accuracy.

Good estimations are characterized by a small value of Mean(RE) [5]. However, it can happen that large positive values of RE_i are balanced by large negative values of RE_i. When this happens, a small value of Mean(RE) may not imply good estimations [5]. Positive values of Mean(RE) indicate that, on average, the remediation time that SonarQube suggests is underestimated, while negative values indicate overestimation. Moreover, the lower the *MdMRE* value is, the better SonarQube's estimations are.

3 RESULTS

As for *Replication 1*, we collected data from 26 projects. However, only 11 projects provided valid data (containing the estimated and the actual effort to fix all the TD items), while for *Replication 2* all of 6 projects provided valid data. The complete results are available in the replication package [4].

3.1 TD items introduced (RQ_1)

Replication 1. The 11 projects violated 107 Sonarqube TD items 4,081 times mainly of type Code Smells (90%). Considering the severity level, 38% of the TD items were Minor, 47% Major, and 11% Critical.

In the analyzed project, only 35 TD items were violated more than 20 times, of which eight TD items were violated more than 100 times. Among these 35 TD items, 30 were Code Smells (CS), three were Bugs, and two were Vulnerability (Vuln.). Considering the severity assigned by SonarQube 18 were Major, 10 Minor, 5 Critical, Critical, and Blocker 1 respectively.

Replication 2. The 6 projects violated 63 SonarQube TD items 497 times. Considering the severity level, 55% were Major, 32% Minor, 4.2% Critical, and 0.4% Blocker.

In the analyzed project, only 12 TD items were violated more than 10 times, of which 7 TD items were violated more than 30 times. Note that we report results only for the TD items violated more than 20 times in all the projects. Among these 12 TD items, 9 were Major, 3 Minor as severity assigned by SonarQube.

Cross-case considerations. Both replications introduced different types of TD Items. Code Smells were the most diffused ones, while security vulnerabilities were very rarely introduced.

> ***Finding 1.*** *Junior developers tend to incur more code smells over other TD types.*

Since the amount of TD items introduced by junior developers is significant, we can proceed with the analysis of the remaining RQ_s.

3.2 TD items commonly refactored (RQ_2)

Replication 1. Considering the TD items violated by developers, out of 4,081 recurrences, 63% was fixed, while the remaining 37% is still open (Table 1). Moreover, among the TD items violated more than 20 times, five of them are fixed in more than 70% of the cases. For three of the 11 projects (prj. #15, prj. #22 and prj. #26) developers did not fix any TD items, while for one project (proj. #10) they fixed all the violated TD items.

Replication 2. Developers fixed all the TD items in their projects. Considering the different analyzed projects, developers spent less the same time to fix the same issues compared with the time estimated by SonarQube. Only in some isolated cases (TD items: *1473*, *1651*, and *1768*) the actual fixing time was higher than the estimated one. For example, TD item *1473* was fixed 56 times, and only in 3 cases belong to the same group the time was near the double (Table 2). All the detailed information is available in the replication package [4].

Cross-case considerations. Developers fixed most of the TD items. While Replication 2 fixed all the introduced TD items, in Replication 1 developers fixed more than 80% of them, mainly focusing on Code Smells. It is interesting to note that Replication 1 fixed all the TD Items classified as Info, even if they did not account for the TD remediation time calculated by SonarQube, as they considered their presence as potentially harmful.

> ***Finding 2.*** *Junior developers tend to refactor more code smells over other TD types.*

3.3 TD items refactoring time (RQ_3)

Replication 1. All the teams dedicated 164 hours (9,856.13 minutes) in total to refactoring activities, with an average of 40 minutes per TD item. 9 projects spent a minimum of 1 minute and a maximum of 2.7 minutes. Two projects (prj. 15 and prj. 18) spent more time to refactor (more than 10 minutes per TD items) even if they were not the most infected ones (93 TD items and 112 TD items respectively). On the contrary, the two projects infected by the highest number of TD items (more than 1,600 TD items), spent the same time to refactor the code. The project (prj. 10) were developers spent less time to refactor (1 minute) did not close any TD items in the code.

Considering Type and Severity, Bug, Code Smells, and Vulnerability are fixed in the same proportion (60 %), while looking at Severity, developers fixed mostly low-level TD items (Info) (88 %) and near 60 % of the other ones. It is important to note that junior developers fixed even if the did not account as TD.

Replication 2. All the junior developer groups dedicated 20 hours (1,217 minutes) in total to refactoring activities within average

Table 1: Refactoring and Fixing Time grouped by Type and Severity (Replication 1) (RQ_1-RQ_4)

TD Items		Diffuseness (RQ_1)	Refactoring (RQ_2)		Refactoring (RQ_3)			Accuracy (RQ_4)		
		#TD introd. items	#TD fixed items	%TD fixed items	overall ref.time	TD fix. time (min)	%TD fix. time (min)	Mean(RE)	MmRE	MdMRE
Type	Bug	122	78	64 %	264	172	65 %	-403%	525%	495%
	CS	3703	2323	63 %	8906	6669	75 %	-167.6%	456%	435%
	Vuln.	1256	162	13 %	6686	504	8 %	-1256%	629%	650%
Severity	Block.	58	30	52 %	113	66	58 %	-374%	436%	445%
	Crit.	465	266	57 %	856	559	65 %	-493%	472%	452%
	Maj.	1945	1245	64 %	5443	4222	78 %	-1898%	581%	584%
	Min.	1557	973	62 %	3343	2405	72 %	-1528%	378%	339%
	Info	56	49	88 %	104	93	89 %	-198%	318%	225%

Table 2: Refactoring and Fixing Time grouped by Type and Severity (Replication 2) (RQ_1-RQ_4)

TD Items		Diffuseness (RQ_1)	Refactoring (RQ_2)		Refactoring (RQ_3)			Accuracy (RQ_4)		
		#TD introd. items	#TD fixed items	%TD fixed items	overall ref.time	TD fix. time (min)	%TD fix. time (min)	Mean(RE)	MmRE	MdMRE
Type	Bug	10	10	100%	15	15	100%	-76%	78%	78%
	CS	475	475	100%	1,225	1,225	100%	-56%	70%	69%
	Vuln.	3	3	100%	3	3	100%	-80%	80%	80%
Severity	Block.	2	2	100%	2	2	100%	70%	70%	70%
	Crit.	21	21	100%	93	93	100%	-63%	69%	68%
	Maj.	312	312	100%	674	674	100%	-64%	63%	63%
	Min.	153	153	100%	473	473	100%	-53%	76%	75%

17.363 minutes per TD item. All the TD items introduced in the projects, differently than in Replication 1, were fixed.

> ***Finding 3***. *Junior developers focus equally on all TD items, independently from the severity and type assigned by SonarQube*

3.4 Remediation time accuracy (RQ_4)

The Remediation time for the vast majority of TD items was lower than the estimated one in both replications (Table 1 and Table 2).

In *Replication 1*, only in 19 TD items, out the 104 fixed (18%), the actual time was higher than the estimated one. On average, the actual remediation time was at least half than the estimated one, while in some cases, even 20 times lower. In *Replication 2*, only in case of a TD Item (TD Item id=1868), and only in a very few instances (18%), developers took longer than expected to fix it.

No noticeable differences emerge between different types of TD items of different severity levels.

The lower remediation time spent by junior developers might be due to their low experience or to the overestimation of the SonarQube remediation time. Developers might have fixed them with a very quick and non-clean solution. We can speculate that senior developers, even if they are expected to be faster to code, might invest more time reasoning on the reasons of the TD Items, and might take longer to think of better solutions to avoid the problems. However, no other TD Items were introduced after the fixing of the previous ones. Therefore, we can assume that they accurately refactored the code.

> ***Finding 4***. *Junior developers spend no more than 50% of the SonarQube estimated remediation time to fix TD Items, in several cases the remediation time is 20 times lower than the estimated one.*

4 THREATS TO VALIDITY

Construct Validity. We adopted the default set of collected measures considered by the sonarQube model since practitioners are reluctant to customize the built-in quality gate and mostly rely on the standard set of rules [28]. Also, we have tried as well as possible to replicate the conditions adopted by practitioners that use this tool, although we are aware that the detection accuracy of some rules may not be precise. Moreover, developers selected the TD items they had to fix to reach the quality gate without adopting any particular prioritization model [11].

Some developers of the considered projects might use SonarQube during software development and thus might remove some TD items while leaving others. This might affect the validity of the results on the diffuseness of TD items. In particular, the presence of TD items (or the presence of some TD items) in some projects might be underestimated. This threat is shared with past work [23, 24] that studied the diffuseness of TD items detected by SonarQube. Therefore, we foster researchers to study the diffuseness of TD items over time so as to understand whether, or not, some TD items are more fixed than others. Our work, besides adding evidence on the diffuseness of TD items, poses the basis for this future research direction.

The participants formed mutually-exclusive teams, each of which worked on a single project. This means that a variation in the results among the projects could be due to the teams, rather than to the projects themselves. Although we gather initial empirical evidence on SonarQube's remediation time, we believe that replications in which more participants are assigned the same TD items in the same projects are needed to strengthen the validity of our results.

Internal Validity. We filtered data and removed all of the data that was not relevant or complete for effort estimation. 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, but we did not remove

such issues from the analysis since we wanted to report the results without modifying the output provided by the tool.

We filtered data and removed all data that were not relevant or complete for effort estimation. 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, but we did not remove such issues when performing the analyses since we wanted to report the results without modifying the output provided by the tool. There might be a learning effect that makes actual remediation time to decrease when a participant fixes several TD items that belong to the same coding rule. Although such an effect should be present in an actual scenario (the more an actual developer fixes TD belonging to given coding rules, the less the spent remediation time should be), it might have some effect on the obtained results.

External Validity. We analyzed a relatively large number of heterogeneous projects. However, we are aware that other projects might lead to slightly different results.

That is to say that our conclusions might not hold when considering a sample of TD items different from ours. For example, we cannot be sure that the overestimation we found for TD items whose effort level is trivial, easy, or medium is confirmed for TD items whose effort level is sizeable, high, or complex. While we gather initial empirical evidence on the SonarQube's remediation time, we highlight the need for further studies focusing, in particular, on a sample of TD items different from ours. The participants in our study where last-year undergraduate students that can be considered novice developers. However, we are aware that the results could be biased by the selection of participants belonging to a set of developers more trained and with more experience in quality assessment tools. However, we believe that senior developers could increase the overestimation of the remediation time, due to their experience. Using students as a proxy for junior developers is subject to great debate in the software engineering community [8, 9].

Reliability. We used standard Python packages to perform all statistical analyses since they ease the replication of the results and increase confidence in their quality.

5 RELATED WORK

Considering TD at level code, the estimation has been evaluated from the general point of view of approaches and strategies [10, 25, 30], and how to measure it, especially considering SonarQube [6, 24, 27]. Another aspect evaluated is the financial aspect of TD [2].

The main works are related to defining approaches that quantify TD in terms of cost to fix technical issues and the interest [21], or that conceptualize the relationship between cost and benefit to improve software quality and help decision-making process during maintenance activities [25].

Another model estimates TD, in particular defect, through the product evolution [1], based on the maintenance cost increases over time due to the code degradation. Another study focused on automated identification of TD comparing this with the human-provided by developers. Results showed a little overlapping between the two estimation [30], while the usage of the tools can help the defect identification [30].

The largest percentage of TD repayment is created by a small subset of issue types [6], and the most frequently introduced TD items are related to low-level coding issues [24]. Only a few works investigated TD estimation based on SonarQube rules, considering the change- and fault-proneness [12, 16, 27]. Previous research highlights that developers are not completely sure about the rules' usefulness [28], [26] provided by SonarQube. Moreover, developers refactor their code according to the high severity level of the identified violations [28] to reduce the risk of faults [26].

These developers' concerns are confirmed also in another study [12] that examined the fault-proneness of SonarQube violations, in order to identify which are actually fault-prone and to assess the fault-prediction model accuracy. Based on an empirical study on 21 well-known mature open-source projects from Apache Software Foundations (ASF), results confirmed that among the 202 SonarQube violations, only 26 have low fault-proneness and violations classified as "bugs" hardly never led to a failure.

Moreover, analyzing the different types and severity of the SonarQube TD items assigned, no significant difference between the clean and infected classes was found [16]. Considering the three different types (*Bugs*, *Code Smells* and *Vulnerabilities*), results showed small effect on change-proneness and no effect on fault-proneness [16].

Considering the fault-proneness, there is no significant difference. Among the TD items that SonarQube claims to increase the fault-proneness (classified as *Bug*), only one out of 36 has a very limited effect. The others never led to failure, and on the contrary resulted to slightly increase the change-proneness [16].

Considering the change-proneness, classes infected by SonarQube violations are more change-proneness than clean classes (not affected by SonarQube violations) [16, 27]. Moreover, the fault-prediction model accuracy was compared with the accuracy if only the 26 violations are considered in the model itself. The accuracy of the current model is extremely low (AUC 50.94%), while the other model is more accurate (AUC 83%).

SonarQube TD prediction was also investigated in order to understand whether its calculated TD could be derived from the other metrics that SonarQube measured and not involved in the computation. Unfortunately, the current software metrics do not predict TD, and that TD does not seem to have a large impact on the lead time to add functionalities and fix bugs [14].

Another study [24] compared the effort needed by developers to repay TD with the SonarQube estimation. The analysis showed that SonarQube remediation time is generally overestimated compared to the actual time for patching TD items. The most accurate estimations are related to *Code Smells*, while the least accurate to *Bugs*.

In recent works [4, 23], the authors analyzed the accuracy of the remediation time estimation associated with TD items, so as to understand the deviation from the prediction, and help companies, but also SonarQube, to better estimate the actual TD. They designed and conducted a case study 1) asking 65 novice developers to remove TD items from fifteen open-source Java projects, and 2) comparing the effort developers needed to remediate TD items with the estimation proposed by SonarQube. The results point out that the remediation time is generally overestimated by the tool as compared to the actual time for remediating TD items, and that

the most accurate estimations relate to *code smells*, while the least accurate concern *bugs*.

In the original study, each team autonomously is chosen, among the TD items identified by SonarQube, which ones had to be fixed to achieve the target values of TD established for the project. In our work, developers should try to fix all the TD items detected by SonarQube.

6 CONCLUSION

In this paper, we present a case study replicated at two universities in two different countries to investigate: (i) the diffuseness of TD items introduced by junior developers, (ii) the TD items commonly refactored by junior developers and (iii) the time commonly spent for refactoring them. Moreover, we also compare the time spent for refactoring TD items (remediation time) with the one estimated by SonarQube and how critical junior developers perceive them.

Results show that developers consistently introduce TD items of different types and severity, with a predominance of TD Items of type "Code Smells" and a very rare introduction of security vulnerabilities. However, when junior developers are asked to refactor TD items, to improve the quality of the code, they address them all, without considering the severities and types.

Another unexpected result was the very low remediation time that our developers took to remove TD items. In previous works [23], remediation time resulted to be overestimated only up to a very low amount, while in this work the remediation time is always overestimated at least by 100%, with some cases that range up to 20 times more. The reasons might be different participants. Differently than in Saarimaki et al. [23], here we only considered students as participants, but with at least two years of full-time professional experience, while in the previous work, students had no professional experience.

On the basis of these findings, we speculate that software companies can be encouraged to adopt SonarQube because of its support in reducing technical debt and they might employ Junior developers to refactor and clean the code efficiently. However, as also recommended by SonarQube, the quality model must be carefully customized by the companies, considering which rules should be included and the related severity based on each context.

There are several future directions for the research presented in this paper. First, we will replicate this study non-academic context, with a different cohort and on larger projects, with different static analysis tools [15], and in different projects [13]. Second, we will better investigate the perceived severity of each rule, so as to understand if there is common agreement on the classification, or if it is completely context-dependent.

REFERENCES

[1] Abdullah Aldaeej and Carolyn Seaman. 2018. From Lasagna to Spaghetti, a Decision Model to Manage Defect Debt. In *Proceedings of the 2018 International Conference on Technical Debt (TechDebt '18)*. 67–71.

[2] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. 2015. The Financial Aspect of Managing Technical Debt. *Inf. Softw. Technol.* 64, C (Aug. 2015), 52–73.

[3] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* 6, 4 (2016), 110–138. https://doi.org/10.4230/DagRep.6.4.110

[4] Maria Teresa Baldassarre, Valentina Lenarduzzi, Simone Romano, and Nyyti Saarimäki. 2020. On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube. *Information and Software Technology* 128 (2020).

[5] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. 1985. Software Effort Estimation and Productivity. Advances in Computers, Vol. 24. 1 – 60.

[6] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou. 2018. How do developers fix issues and pay back technical debt in the Apache ecosystem? *International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[7] Eric Evans. 2004. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[8] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical Software Engineering Experts on the Use of Students and Professionals in Experiments. *Empirical Softw. Engg.* 23, 1 (2018), 452–489.

[9] Robert Feldt, Thomas Zimmermann, Gunnar R. Bergersen, Davide Falessi, Andreas Jedlitschka, Natalia Juristo, Jürgen Münch, Markku Oivo, Per Runeson, Martin Shepperd, Dag I. SjØberg, and Burak Turhan. 2018. Four Commentaries on the Use of Students and Professionals in Empirical Software Engineering Experiments. *Empirical Softw. Engg.* 23, 6 (2018), 3801–3820.

[10] Y. Guo, R. Spínola, and C. Seaman. 2016. Exploring the costs of technical debt management – a case study. *Empirical Software Engineering* 21, 1 (2016), 159–182.

[11] Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, and Francesca Arcelli Fontana. 2020. Technical Debt Prioritization: State of the Art. A Systematic Literature Review. arXiv:cs.SE/1904.12538

[12] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi. 2020. Are SonarQube Rules Inducing Bugs?. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER2020)*. 501–511.

[13] Valentina Lenarduzzi, Francesco Lomio, Nyyti Saarimäki, and Davide Taibi. 2020. Does migrating a monolithic system to microservices decrease the technical debt? *Journal of Systems and Software* 169 (2020).

[14] Valentina Lenarduzzi, Antonio Martini, Davide Taibi, and Damian Andrew Tamburri. 2019. Towards Surgically-precise Technical Debt Estimation: Early Results and Research Roadmap. In *International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE'19)*. 37–42.

[15] Valentina Lenarduzzi, Vili Nikkola, Nyyti Saarimäki, and Davide Taibi. 2020. Does code quality affect pull request acceptance? An empirical study. *Journal of Systems and Software* (2020).

[16] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2019. Some SonarQube Issues have a Significant but Small Effect on Faults and Changes. A large-scale empirical study. *arXiv:1908.11590* (2019).

[17] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. 2017. Analyzing Forty Years of Software Maintenance Models. In *39th International Conference on Software Engineering Companion (ICSE-C '17)*. 146–148.

[18] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. 2020. A Survey on Code Analysis Tools for Software Maintenance Prediction. In *6th International Conference in Software Engineering for Defence Applications*.

[19] Vladimir Mandic, Nebojša Taušan, and Robert Ramac. 2020. The Prevalence of the Technical Debt Concept in Serbian IT Industry: Results of a National-Wide Survey. In *International Conference on Technical Debt*. 10.

[20] Sam Newman. 2015. *Building microservices: designing fine-grained systems* (1st ed.). O'Reilly Media, Inc.

[21] A. Nugroho, J. Visser, and T. Kuipers. 2011. An Empirical Model of Technical Debt and Interest. In *Workshop on Managing Technical Debt (MTD '11)*. 1–8.

[22] P. Runeson and M. Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Softw. Engg.* 14, 2 (2009), 34.

[23] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, and S. Romano. 2019. On the Accuracy of SonarQube Technical Debt Remediation Time. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2019)*. 317–324.

[24] Nyyti Saarimäki, Valentina Lenarduzzi, and Davide Taibi. 2019. On the Diffuseness of Code Technical Debt in Java Projects of the Apache Ecosystem. In *Second International Conference on Technical Debt (TechDebt '19)*. 98–107.

[25] Carolyn Seaman and Yuepu Guo. 2011. Measuring and Monitoring Technical Debt. *Advances in Computers* 82 (2011), 25 – 46.

[26] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. 2017. How developers perceive smells in source code: A replicated study. *Information and Software Technology* 92, Supplement C (2017), 223 – 235.

[27] I. Tollin, F. Arcelli Fontana, M. Zanoni, and R. Roveda. 2017. Change Prediction Through Coding Rules Violations. In *21st International Conference on Evaluation and Assessment in Software Engineering (EASE'17)*. 61–64.

[28] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. *International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)* (2018), 38–49.

[29] R.K. Yin. 2018. *Case Study Research: Design and Methods* (6th ed.). SAGE.

[30] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. 2013. A Case Study on Effectively Identifying Technical Debt. In *International Conference on Evaluation and Assessment in Software Engineering (EASE '13)*. 42–47.