On the Accuracy of SonarQube Technical Debt Remediation Time

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

Maria Teresa Baldassarre
University of Bari
Bari, Italy
mariateresa.baldassarre@uniba.it

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

Simone Romano
University of Bari
Bari, Italy
simone.romano@uniba.it

***Abstract*—[Context] The popularity of tools for software quality analysis has increased over the years, with special attention to tools that calculate technical debt based on violations of a set of rules. SonarQube is one of the most used tools and provides an estimation of the time needed to remediate technical debt. However, practitioners are still skeptical about the accuracy of its remediation time estimation. [Objective] In this paper, we analyze the accuracy of SonarQube remediation time on a set of 15 open source Java projects. [Method] We designed and conducted a case study where we asked 65 novice developers to remove rule violations and reduce the technical debt of 15 projects. [Results] The results point out that SonarQube remediation time, compared to the actual time for reducing technical debt, is generally overestimated, and that the most accurate estimation relates to code smells, while the least accurate concerns bugs. [Conclusions] Practitioners and researchers could benefit from the results of this work to understand up to which extent technical debt is overestimated and have a more accurate estimation of the remediation time.**

***Index Terms*—Technical debt, remediation time, effort estimation, code smells**

I. Introduction

The popularity of tools for measuring internal software quality is rapidly increasing [1] [2]. Among the existing static analysis tools, SonarQube has been adopted by more than 100K organizations[1] including nearly more than 15K public open-source projects.[2]

SonarQube checks code compliance against a set of coding rules. If the code violates any of the classified rules, SonarQube considers it as a violation or a *Technical Debt* (TD) item. SonarQube calls the estimated effort needed to refactor the violated rule as *remediation time* and considers it as TD.

In this work, we aim to understand the accuracy of the remediation time estimation associated to TD items, so as to understand the deviation from the prediction, and help companies, but also SonarQube, to better estimate the actual TD. For this purpose, we designed and conducted a case study where we asked 65 novice developers to remove TD items from fifteen open source Java projects. We then compared 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*.

The remaining part of the paper is structured as follows. Section II describes some related work done by researchers in the last years. In Section III, we present the planning and execution of our case study. In Section IV, we show the obtained results, which we discuss in Section V. Section VI focuses on limitations and how we addressed threats to the validity of our study. In Section VII, we draw conclusions and outline some possible future work.

II. Related Work

Different approaches and strategies have been proposed to evaluate TD. At the best of our knowledge, no work has yet investigated the accuracy of the TD remediation time of SonarQube rules. In this section, we introduce related work on TD estimation.

Nugroho et al. [3] proposed an approach to quantify TD in terms of cost to fix technical issues and its interest. They monitored data from 44 systems. They empirically validated the approach on a real system.

Seaman et al. [4] proposed a TD management framework that formalizes the relationship between cost and benefit in order to improve software quality and help decision-making process during maintenance activities.

Guo et al. [5] investigated the TD cost of applying a new approach on an on-going software project. They found a higher start-up cost that decreased over time.

Different approaches or strategies have been proposed to manage TD.

Guo et al. [6] proposed a portfolio approach in order to help the software manager in decision making. This approach provides a new perspective for TD management.

Nord et al. [7] defined a measurement-based approach to develop metrics in order to strategically manage TD. This approach could optimize the development cost over time without stopping the development process. They successfully applied the approach to an ongoing system development effort.

Saarimäki et al. [8] investigated the diffuseness of TD items in Java projects, reporting that the most frequently introduced TD items are related to low-level coding issues. Also in their case, they did not consider the TD remediation time.

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

III. Case Study Design

We designed our empirical study as a case study based on the guidelines defined by Runeson and Höst [9]. We have made the raw data available on the web, in order to make the study replicable.[3]

A. Goal and Research Questions

The goal of this work is to evaluate the accuracy of the TD remediation time of SonarQube, by comparing the actual with the estimated remediation time. Accordingly, we formulated our goal as follows, using the Goal/Question/Metric (GQM) template [10]:

Purpose	analyze
Object	TD items
Quality	with respect to their remediation time accuracy
Viewpoint	from the point of view of novice developers
Context	in the context of Java projects

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

RQ1. What is the diffuseness of TD items in source code?
RQ2. What is the accuracy of the TD remediation time?
RQ3. What is the accuracy of the TD remediation time considering different types and levels of severity?

With RQ1 we aim to determine the extent to which TD items are prevalent in the considered open-source projects. When studying RQ1, we also took into account the diffuseness of TD items with respect to their type—i.e., *Bugs*, *Vulnerabilities*, and *Code Smells*—and levels of severity—i.e., *Info*, *Minor*, *Major*, *Critical*, and *Blocker*. The results from this RQ would also help us to better understand the context of our study.

As for RQ2, we aim to evaluate the accuracy of the TD remediation time SonarQube suggests by comparing the actual with the estimated remediation time. For this RQ, we considered the remediation time of all TD items.

Finally, with RQ3 we studied the TD remediation time with respect to the type and severity level of TD items. This is to determine if the accuracy of the remediation time is consistent among different severity levels and types. For both RQ2 and RQ3, we applied different accuracy evaluation criteria in order to obtain more complete results.

B. Context

We considered a set of projects written in Java, which satisfied the following criteria: *(i)* hosted on GitHub; *(ii)* size greater than 10KLOC (i.e., 10,000 Lines Of Code); *(iii)* SonarQube's rating worse than "A" for at least two out of three TD types (i.e., bugs, vulnerabilities, and code smells); and *(iv)* presence of a regression test suite. The rationale behind the criteria *(i)* and *(ii)* was to select real open-source projects with a non-trivial size. Criterion *(iii)* aimed to discard projects with few TD items so as to force the participants in our study to spend a significant amount of time to remedy TD items. Finally, the last criterion allowed having a safety net when remediating TD items. Main information on the considered projects is summarized in Table I. By looking at the table, we can grasp to what extent the remediation time, estimated by SonarQube, varied among the projects. Despite a larger estimated remediation time for the code smell type, with respect to the bugs and vulnerabilities types, we can notice that SonarQube's rating is "A" for all the projects. This is because SonarQube rates the type code smell differently from how it rates the other two types. Based on the lines of code, number of classes (i.e., #Classes), and McCabe's Cyclomatic Complexity [11] (i.e., CC) values, we can also observe the different complexity of the considered projects. Finally, Table I also highlights the size of the regression test suite for each project (i.e., #Tests). Summing up, we can conclude that the considered projects are heterogeneous and therefore suitable for the study.

Our study was run within the Computer Science program at the University of Bari (Bary, Italy). The participants were third-year undergraduate students who were taking the *Software Quality* course. This course included both face-to-face and laboratory lessons, and covered the following topics: software quality (i.e., internal, external, and in-use); ISO standards for software quality; software quality assessment, monitoring, and improvement [12], [13]; and supporting tools for quality management (e.g., SonarQube) and process control [14]. The students had to carry out project work on an assignment by working in teams, consisting in improving the quality of a Java project in terms of reliability, security, and maintainability perspectives. These perspectives relate to the bugs, vulnerabilities, code smells types, respectively. For example, to improve the reliability of a project, bugs should be removed; to improve its security, vulnerabilities should be removed; and so on.

The students' background was quite homogeneous based on some demographic information we gathered, by means of a questionnaire, at the beginning of the course. In particular, all but one student, involved in this study, had passed the exam on object-oriented programming, with high marks (i.e. grades of 75%). Moreover, their self-reported experience in Java was median of 4 on a 5-point scale from "very inexperience" (1) to "very experienced" (5). The sample of 65 students is also representative of novice developers, given their background and considering they were all close to graduating and most likely would be working within a range of six months.

C. Procedure and Data Collection

To measure the accuracy of TD remediation time, we needed to collect the actual time it took to remedy TD items. To this end, we used the assignments we gave to the students of the Software Quality course. In particular, we defined and put into practice the following protocol:

1) The students had to form (mutually-exclusive) teams, by randomly assigning them. These teams ranged from two to five participants each and had a team leader, elected by the team members.

[3] https://github.com/clowee/TechnicalDebtRemediationTime

TABLE I: Some information on the selected projects.

Project (GitHub page)	Estimated Remediation Time (Rating)			KLOC	#Classes	CC	#Tests
	Bug	Vulnerability	Code Smell				
Apache PDFBox (github.com/apache/pdfbox)	1d (E)	2d (D)	51d (A)	135	1,274	22,353	1,635
Computoser (github.com/Glamdring/computoser)	4h 55m (E)	7h 30m (E)	10d (A)	12	157	2,953	12
Flickr4Java (github.com/boncey/Flickr4Java)	1h 30m (E)	6h 45m (B)	15d (A)	14	166	2,957	135
GameComposer (github.com/mirkosertic/GameComposer)	3h 52m (E)	2d (B)	59d (A)	22	469	5,451	233
IRI (github.com/iotaledger/iri)	3h 55m (E)	1d 1h (E)	12d (A)	10	162	2,094	202
jChecs (github.com/aapiro/JChecs)	3h 45m (E)	1h 10m (B)	13d (A)	10	103	2,035	41
jsoniter (github.com/json-iterator/java)	6h 5m (C)	1d 2h (B)	12d (A)	13	174	2,761	1,337
Libresonic (github.com/Libresonic/libresonic)	2d 2h (E)	2d 1h (E)	31d (A)	30	354	9,866	92
MyBatis (github.com/mybatis/mybatis-3)	3h 35m (E)	1h 5m (B)	19d (A)	22	391	4,391	1,464
Ninja (github.com/ninjaframework/ninja)	7h 26m (E)	1d 2h (E)	16d (A)	18	425	2,983	1,012
OkHttp (github.com/square/okhttp)	1d 4h (E)	6h 51m (E)	18d (A)	24	295	5,329	2,581
OpenAudible (github.com/openaudible/openaudible)	1d (E)	2d 7h (B)	23d (A)	15	163	3,281	13
RoaringBitmap (github.com/RoaringBitmap/RoaringBitmap)	7h (E)	6h 45m (B)	36d (A)	28	246	6,126	4,547
Traccar (github.com/traccar/traccar)	7h 30m (E)	40m (E)	25d (A)	47	697	8,267	340
TrackMate (github.com/fiji/TrackMate)	1d 4h (D)	4d 4h (B)	70d (A)	46	366	6,529	69

* Estimated remediation times, ratings, KLOC, #Classes, and CC are computed through SonarQube. #tests are gathered by means of Apache Maven.
† When reporting estimated remediation times, d, h, and m stand for days, hours, and minutes, respectively.

2) Each team searched for a project on GitHub, which had to satisfy the criteria from *(i)* to *(iv)* (see Section III-B). When the team found a project satisfying those criteria, the team leader sent a project proposal to the authors MB and RS, who decided whether to approve the project or not. The proposal included a SonarQube report, which allowed MB and RS to evaluate if the criteria were satisfied. Furthermore, they assigned target values for reducing TD, based on project characteristics such as size, severity and, number of items, as well as team size, i.e. projects had to achieve a triple *A* value for these characteristics and a TD of at most 2-3 days.
3) Once the project was approved, the team defined an "action plan", in other words, team members autonomously identified the set of SonarQube items they chose to solve (i.e. what we indicate from here on as fixed TD items). The goal of the action plan was to reduce TD with respect to reliability (bugs), security (vulnerabilities) and maintainability (code smells) perspectives. As TD items were addressed, the internal software quality was iteratively monitored through SonarQube status reports, until the target thresholds established for every project were finally achieved. When fixing TD items, the students were supported by an application life-cycle management tool (i.e., Redmine[4]). Thanks to this tool they kept track of the actual time needed to remediate TD items by using appropriate tags. This allowed us to assess differences (if any) between the estimated and actual remediation time for addressing TD issues.
4) For each fixed TD item, we extracted information on the Sonar TD *squid*—i.e. an identifier for TD items—, actual and estimated remediation time, item type (i.e., bug, vulnerability, or code smells) and severity level (i.e., info, minor, major, critical, or blocker).

[4]https://www.redmine.org/

D. Data Analysis

In order to examine the effort estimation accuracy, we applied six accuracy evaluation criteria: *RE* (Relative Error), *MMRE* (Mean Magnitude Relative Error), *MdMRE* (Median Magnitude Relative Error), *MAR* (Mean Absolute Residual), *PRED25*, and *PRED50*. The usage of different indicators allows us to obtain a more complete picture of accuracy.

MAE is defined as follow:

$$RE = \frac{\text{actual\_effort} - \text{estimated\_effort}}{\text{estimated\_effort}}$$

MRE is the absolute value of RE and both MMRE and MdMRE are based on MRE [15]. It is the magnitude of relative error calculated as follows:

$$MRE = \frac{|\text{actual\_effort} - \text{estimated\_effort}|}{\text{estimated\_effort}}$$

MRE expresses how many percentages the estimated effort was different from the actual effort. For example, MRE equal to 0.3 means that the relative error of prediction is 30%. MMRE is the mean of MRE-values. As a mean, MMRE could be strongly influenced by few very high MRE values [16]. For this reason, we also calculated the median of MRE values, namely MdMRE. MMRE is an asymmetric measure, which means it could be a biased estimator of central tendency of the residuals of a prediction system [17], [18]. To avoid this bias, Shepperd and MacDonell [19] proposed estimating the accuracy using MAR:

$$MAR = \sum_{i=1}^{n} \frac{|\text{actual\_effort}_i - \text{estimated\_effort}_i|}{n},$$

where n is the number of observations. Moreover, we considered PRED25 and PRED50. PRED25 is the percentage of the observations with MRE $\leq$ 0.25, while PRED50 is the percentage of the observations with MRE $\leq$ 0.50.

IV. Results

In this section, we report the results with respect to our RQs.

TABLE II: TD item diffuseness for each project considering the total number and the different severity levels and types (RQ1).

Project name	Severity					Type			Total
	Info	Minor	Major	Critical	Blocker	Bug	Vulnerability	Code Smell	
Apache PDFBox	288	718	966	621	61	132	106	2,416	2,654
Computoser	37	160	293	70	5	41	33	491	565
Flickr4Java	91	934	388	206	5	14	40	1,570	1,624
GameComposer	11	683	373	234	77	32	85	1,261	1,378
IRI	7	252	242	82	12	22	54	519	595
jChecs	2	227	178	383	22	54	10	748	812
jsoniter	4	262	210	121	1	20	84	494	598
Libresonic	18	283	630	218	26	60	31	1,084	1,175
MyBatis	28	469	857	385	16	160	32	1,563	1,755
Ninja	66	490	469	191	15	58	119	1,054	1,231
OkHttp	36	359	581	214	48	69	36	1,133	1,238
OpenAudible	50	655	730	450	0	44	192	1,649	1,885
RoaringBitmap	110	620	601	250	84	72	60	1,533	1,665
Traccar	6	157	374	332	16	98	23	764	885
TrackMate	32	1,868	841	511	146	84	179	3,135	3,398
All projects	786	8,137	7,733	4,268	534	960	1,084	19,414	21,458

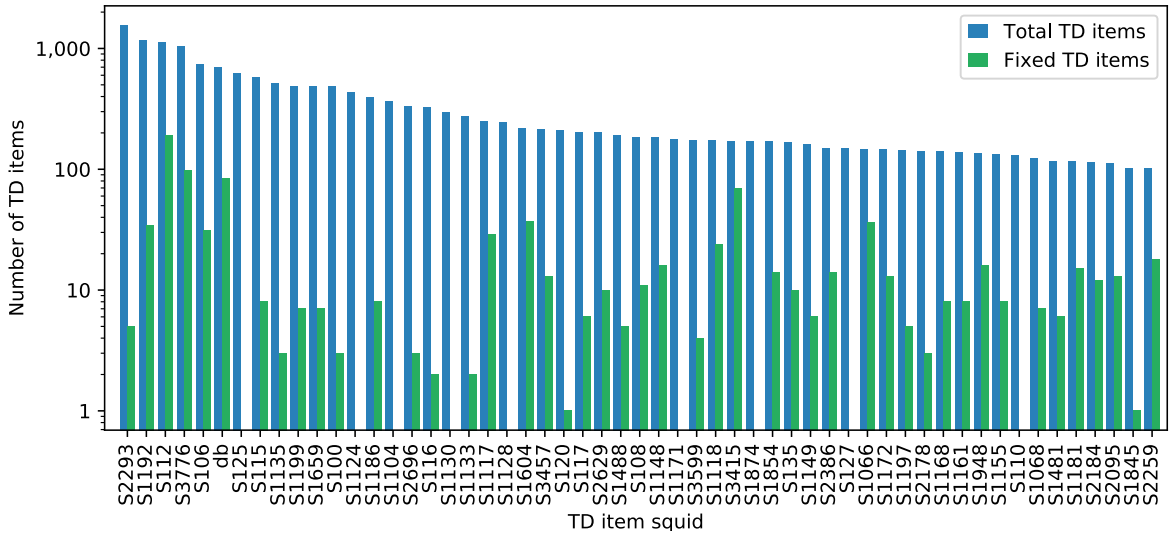

Fig. 1: Distribution of TD items with more than 450 introductions in the analyzed projects (RQ1).

A. *(RQ1). What is the diffuseness of TD items in source code?*

Out of the 1,568 TD items classified by SonarQube, 243 were detected in the set of 15 analyzed projects. They were introduced in the systems 21,458 times. Table II shows how the TD items are distributed in each project with respect to severity levels and type. In Figure 1, for space reasons, we only reported the TD items detected more than 450 times in the analyzed projects. Here, TD items on the x-axis are rule identifiers (i.e., squid) from SonarQube, except for the item *db*, which is an abbreviation of the rule called *common-java:DuplicatedBlocks*. In addition, the y-axis of the figure uses a logarithmic scale in order to visualize the number of fixed TD items in the same figure. We can see that *S2293* is the most recurrent TD item with 1,560 introductions, while *S1192*, *S112*, *S3776* have a frequency higher than one thousand.

Figure 1 also contains the number of fixed TD items for each TD item. The most fixed ones were S112, S3776, and db with 191, 98, and 84 fixes, respectively. The number of fixed items is significantly smaller than the number of total TD items for all of the most common TD items.

Moreover, we grouped the TD items by project and then by severity level and type (Figure 2). By considering the severity level, the large majority of projects have most of their TD items classified as minor, major, or critical, while info and *blocker* constitute only 6% of the total introductions. When considering the type of the TD items, almost all of them are code smells (19,414 out of 21,458), while vulnerability and bug represent a low percentage of the total.

Finally, we identified the most recurrent TD items in the analyzed projects. In particular, we reported in Table III the top 12 TD items, which occur (in total) 9,487 times. The distribution of the most common TD items in the analyzed projects is presented in Figure 3. Most of the distributions are right-skewed, which is true especially for the two most common TD items. For all of the TD items, the medians are larger than zero, indicating that such TD items are present in most of the analyzed projects.

The interested reader can find in Table VIII (at the end of

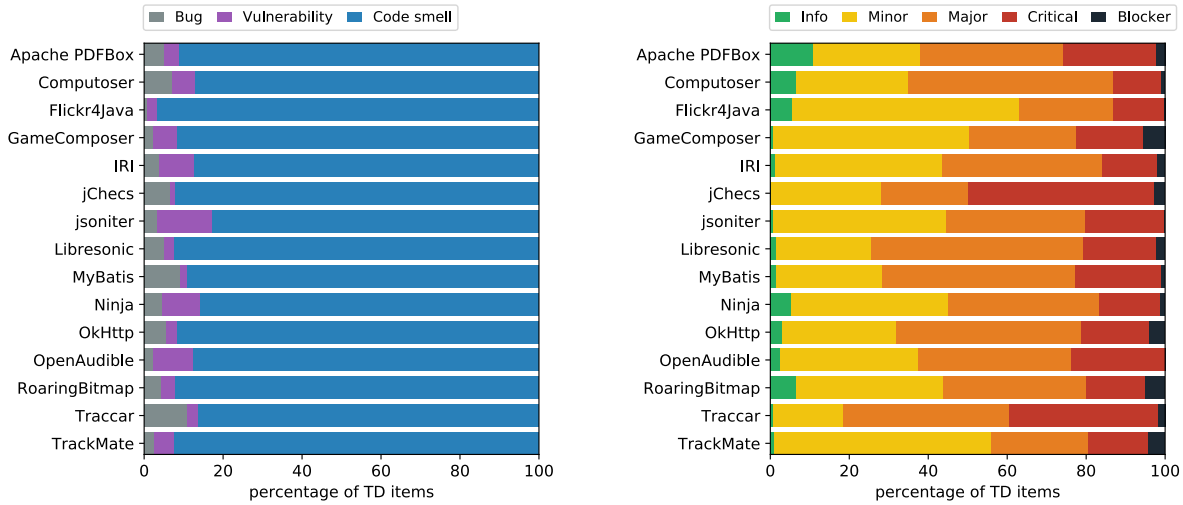

Fig. 2: Average contribution of TD items in the analyzed projects with respect to the severity level and type of the items (RQ1).

TABLE III: Top 12 TD items identified in the analyzed projects (RQ1).

Project name	**TD item**												**Total**
	S2293	S1192	S112	S3776	S106	db	S125	S115	S1135	S1659	S1199	S100	
Apache PDFBox	2	138	48	248	250	92	27	105	214	50	62	1	1,237
Computoser	35	17	10	39	68	5	25	0	33	2	0	1	235
Flickr4Java	546	133	7	8	6	39	17	9	21	28	0	14	828
GameComposer	3	31	13	34	2	19	39	32	11	96	121	87	488
IRI	5	8	64	19	4	7	25	7	7	40	30	0	216
jChecs	0	8	0	88	10	19	3	5	2	0	1	0	136
jsoniter	0	36	13	49	8	34	4	2	4	1	0	6	157
Libresonic	167	131	270	42	25	23	13	1	16	1	0	0	689
MyBatis	4	284	308	37	12	180	19	6	8	1	33	163	1,055
Ninja	17	18	86	26	21	5	27	34	5	0	0	18	257
OkHttp	0	36	47	50	113	8	32	35	30	2	0	0	353
OpenAudible	8	50	51	34	58	10	243	229	50	35	10	20	798
RoaringBitmap	45	36	162	124	22	67	68	21	76	142	4	163	930
Traccar	1	177	40	138	0	76	4	0	6	15	0	0	457
TrackMate	727	65	3	98	139	114	71	93	32	74	226	9	1,651
All projects	1,560	1,168	1,122	1,034	738	698	617	579	515	487	487	482	9,487

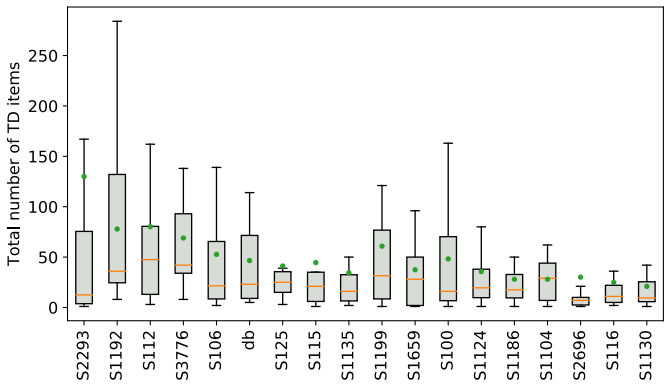

Fig. 3: Distribution of TD items introduced at least 300 times in the analyzed projects (RQ1)

the paper) a short description of the top 12 TD items (i.e., the rule these TD items violate) as reported by SonarQube, as well as their type and severity level. Further details and descriptions for all the TD items monitored by SonarQube can be found in the replication package.

B. (RQ2). What is the accuracy of the TD remediation time?

Table IV reports the diffuseness of the fixed TD items for each project, while Table V presents the results related to the TD remediation time considering all the fixed TD items in each project. For each project, we applied the five accuracy evaluation criteria (MMRE, MdMRE, MAR, PRED25, and PRED50) as reported in Section III-D.

Out of 21,458 TD items affecting the projects (Table II, column "Total"), 1,637 were fixed by the participants in our study (Table IV, column "Total").

Looking at the estimation accuracy (Table IV), only two projects (i.e., RoaringBitmap and MyBatis) have a good accuracy with MMRE values of 0.35 and 0.47 and MdMRE values equal to 0 and 0.42. The results are also confirmed by MAR. Considering PRED25, we can see that a good percentage (53% as maximum) has accuracy values less or equal to 0.25. However, considering RE, the remediation time is generally underestimated for six projects (this is because the sign of RE

TABLE IV: Diffuseness of the fixed TD items for each project with respect to the type and severity level (RQ2 and RQ3).

Project name	Severity					Type			Total
	Info	Minor	Major	Critical	Blocker	Bug	Vulnerability	Code Smell	
Apache PDFBox	47	46	4	6	0	14	9	80	103
Computoser	0	10	16	12	2	11	6	23	40
Flickr4Java	2	30	26	5	2	4	5	56	65
GameComposer	0	14	8	1	1	8	12	4	24
IRI	0	23	30	10	2	6	5	54	65
jChecs	2	54	63	84	12	24	4	187	215
jsoniter	0	26	38	18	0	9	3	70	82
Libresonic	0	7	10	2	2	14	5	2	21
MyBatis	1	13	94	39	3	13	5	132	150
Ninja	0	93	65	12	15	27	57	101	185
OkHttp	0	6	18	2	3	10	4	15	29
OpenAudible	0	16	23	12	1	5	6	41	52
RoaringBitmap	0	0	496	0	0	0	0	496	496
Traccar	0	14	29	6	7	6	2	48	56
TrackMate	0	28	21	4	1	8	6	40	54
All projects	52	380	941	213	51	159	129	1,349	1,637

TABLE V: TD remediation time accuracy (RQ2).

Project name	All fixed TD items					
	RE (mean)	MMRE	MdMRE	MAR	PRED25	PRED50
ApachePDFBox	0.27	0.92	0.62	4.03	25%	42%
Computoser	-0.20	0.90	0.74	1.15	16%	30%
Flickr4Java	-0.09	0.97	0.75	1.76	17%	25%
GameComposer	0.87	1.81	0.71	2.12	13%	33%
IRI	-0.81	0.92	1.00	1.35	8%	11%
jChecs	0.87	1.35	0.33	0.24	46%	63%
jsoniter	-0.30	0.95	0.76	0.89	12%	19%
Libresonic	1.95	2.55	0.91	6.67	14%	22%
MyBatis	-0.36	0.47	0.42	0.62	40%	53%
Ninja	-0.20	0.91	0.75	0.95	16%	27%
OkHttp	0.67	1.18	0.50	3.33	18%	46%
OpenAudible	-0.50	0.75	0.73	2.22	10%	20%
RoaringBitmap	-0.22	0.35	0.00	0.17	53%	63%
Traccar	0.17	0.85	0.50	4.80	34%	48%
TrackMate	-0.34	0.56	0.57	3.57	27%	41%
All projects	-0.01	0.80	0.59	1.28	33%	45%

is positive) and overestimated for the remaining nine (this is because the sign of RE is negative).

C. (RQ3). What is the accuracy of the TD remediation time considering different types and levels of severity?

Among others, Table IV also reports the fixed TD item diffuseness for each project considering the different types and severity levels. As highlighted in RQ2, developers ultimately needed to fix a small portion of the TD items to achieve the set thresholds. Considering the TD types, developers fixed less than 20% (16% of the bugs, 12% of the vulnerabilities and 7% of the code smells affecting the projects). The same trend was found looking at the TD severity level (12% of major, 9% of blocker, 6% of info, 5% of critical and 4.6% of minor). In Table VI, we reported the results grouped by TD type, namely Code Smell (CS), Vulnerability (V), and Bug (B). The remediation time estimates for the type code smell were more accurate than the estimates for the types vulnerability and bug (MMRE, MdMRE, and MAR). With respect to the projects, the estimations were accurate for jChecs, RoaringBitmap, MyBatis and Traccar. However, considering the RE value we can see again that for the vast majority (more than 70%) of the projects the estimated effort to fix the TD items was overestimated by SonarQube. Indeed, this trend is evident when considering the types code smell and vulnerability while it is not evident for the bug type. Considering PRED25 we can see that a good percentage (50% on average) had accuracy values less or equal to 0.25.

Table VII contains the results grouped by severity level. We report the results only for MAjor (MA) and MInor (MI), as we did not have enough data to report the results for levels info, critical and blocker.

In terms of severity type, we cannot see a substantial difference in the estimation accuracy. The effort tends to be overestimated for both major and minor severity levels. The remediation time estimation was especially good for the projects Flickr4JAva, jChecs, MyBatis, and RoaringBitmap. Considering PRED25, we can see that a good percentage (50% on average) has accuracy values minor or equal to 0.25.

V. Discussion

The TD items fixed in the selected projects were removed with less effort than expected. The estimated remediation time was commonly higher than the time spent by our participants to remove the TD items. It is important to notice that in our case study, the participants were students, and that they

TABLE VI: TD remediation time accuracy for the fixed TD items with respect to their type, namely: Code Smell (CS), Vulnerability (V), and Bug (B) (RQ3).

Project name	RE (mean)			MMRE			MdMRE			MAR			PRED25			PRED50		
	CS	V	B	CS	V	B	CS	V	B	CS	V	B	CS	V	B	CS	V	B
Apache PDFBox	0.28	0.00	0.42	0.54	1.00	1.06	0.54	0.78	0.88	4.73	3.78	0.61	27%	11%	14%	46%	22%	29%
Computoser	-0.54	-0.73	0.97	0.75	0.73	1.59	0.75	0.75	0.49	1.75	0.85	0.37	17%	0%	18%	22%	17%	55%
Flickr4Java	-0.05	-0.18	-	0.78	0.18	-	0.78	0.00	-	1.95	0.47	-	11%	80%	-	20%	80%	-
GameComposer	-	-0.61	3.99	-	0.61	4.07	-	0.71	0.46	-	0.93	0.33	-	17%	13%	-	25%	63%
IRI	-0.84	-1.00	-0.38	1.00	1.00	0.79	1.00	1.00	1.00	1.37	1.93	0.65	9%	0%	0%	11%	0%	17%
jChecs	0.00	-	7.84	0.25	-	8.06	0.25	-	0.42	0.18	-	0.69	48%	-	33%	65%	-	54%
jsoniter	-0.14	-	-0.26	0.73	-	0.75	0.73	-	0.66	1.07	-	0.18	14%	-	22%	23%	-	22%
Libresonic	-	2.81	3.86	-	2.95	4.31	-	1.00	1.00	-	1.11	1.34	-	0%	14%	-	40%	14%
MyBatis	-0.37	-0.32	-0.24	0.39	0.32	0.48	0.39	0.00	0.60	0.69	0.29	0.14	39%	60%	46%	54%	60%	46%
Ninja	-0.29	-0.04	0.41	0.73	0.96	1.23	0.73	0.82	1.00	1.72	0.36	0.24	17%	14%	19%	30%	19%	30%
OkHttp	-0.10	-	0.34	0.62	-	0.69	0.62	-	0.29	5.62	-	0.83	7%	-	40%	40%	-	70%
OpenAudible	-0.68	-0.84	-0.39	0.75	0.84	0.39	0.75	0.87	0.33	2.68	2.82	0.43	2%	0%	40%	10%	0%	60%
RoaringBitmap	-0.22	-	-	0.00	-	-	0.00	-	-	0.17	-	-	53%	-	-	63%	-	-
Traccar	-0.18	-	2.04	0.43	-	2.04	0.43	-	0.65	1.72	-	1.16	40%	-	17%	54%	-	33%
TrackMate	-0.32	-0.49	-0.42	0.60	0.49	0.52	0.60	0.63	0.45	4.37	3.45	0.84	25%	33%	13%	35%	33%	50%
All projects	-0.22	-0.07	1.91	0.50	0.96	2.50	0.50	0.75	0.67	1.27	1.10	0.56	37%	17%	23%	49%	24%	40%

TABLE VII: TD remediation time accuracy of the fixed TD items with respect to the severity level, namely: MInor (MI) and MAjor (MA) (RQ3).

Project name	RE (mean)		MMRE		MdMRE		MAR		PRED25		PRED50	
	MI	MA	MI	MA	MI	MA	MI	MA	MI	MA	MI	MA
Apache PDFBox	0.28	0.08	0.91	0.82	0.67	0.52	2.51	6.47	19%	33%	40%	47%
Computoser	-0.61	-0.68	0.61	0.73	0.67	0.75	0.55	2.26	10%	0%	30%	13%
Flickr4Java	-0.35	-0.44	0.50	0.84	0.55	0.87	0.47	1.20	33%	0%	40%	12%
GameComposer	-0.47	2.59	0.58	3.58	0.63	0.82	0.85	4.81	14%	13%	36%	38%
IRI	-0.66	-0.93	0.88	0.95	1.00	1.00	1.31	1.21	17%	0%	17%	7%
jChecs	0.08	-0.10	0.27	0.46	0.00	0.34	0.02	0.29	77%	41%	83%	68%
jsoniter	-0.44	0.11	0.71	1.34	0.69	0.76	1.06	1.09	15%	13%	23%	16%
Libresonic	1.87	2.36	2.15	2.98	1.00	0.83	1.19	14.85	0%	20%	14%	20%
MyBatis	0.13	-0.44	0.50	0.52	0.50	0.62	0.24	0.89	46%	37%	46%	45%
Ninja	-0.06	-0.35	0.95	0.85	0.75	0.80	0.74	1.61	18%	14%	28%	28%
OkHttp	0.97	-0.04	1.49	0.59	1.20	0.50	7.56	2.35	0%	28%	33%	44%
OpenAudible	-0.74	-0.75	0.74	0.75	0.73	0.81	1.86	3.47	0%	4%	6%	13%
RoaringBitmap	-	-0.22	-	0.35	-	0.00	-	0.17	-	53%	-	63%
Traccar	0.66	-0.13	0.98	0.56	0.13	0.50	0.28	2.26	50%	35%	64%	48%
TrackMate	-0.18	-0.49	0.60	0.52	0.50	0.61	2.91	4.21	29%	24%	39%	38%
All projects	-0.06	-0.21	0.76	0.58	0.63	0.50	1.18	1.20	29%	39%	39%	50%

refactored code of open source projects that they had not developed. This surely enforces and confirms the obtained results. We believe that experienced developers, and especially the original developers, would have spent even less time to remove the same TD issues since they would not need to spend time to comprehend the source code. Unexpectedly, it was enough to remove a significantly small number of TD items to reach the fixed thresholds that we had set for each project. Developers removed on average 7.6% of the total TD items. This was probably due to the fact that developers mainly fixed TD items with the major severity level. We are aware that the vast majority of fixed TD items were classified as code smell. In our case, students were free to choose which TD items they wanted to remove, and the removal of vulnerabilities commonly required much more time, and in some case major refactorings. Another unexpected result, besides the overestimation of the remediation time for all of the TD items, is that the most accurate estimations were related to code smell, while the least accurate ones concerned bugs. We did not expect any differences in this sense.

TABLE VIII: A description for the top 12 TD items together with their type and severity level.

TD item	Description	Type	Severity
S2293	The diamond operator (“<>”) should be used.	Code smell	Minor
S1192	String literals should not be duplicated.	Code smell	Critical
S112	Generic exceptions should never be thrown.	Code smell	Major
S3776	Cognitive Complexity of methods should not be too high.	Code smell	Critical
S106	Standard outputs should not be used directly to log anything.	Code smell	Major
db	Source files should not have any duplicated blocks.	Code smell	Major
S125	Sections of code should not be commented out.	Code smell	Major
S115	Constant names should comply with a naming convention.	Code smell	Critical
S1135	Track uses of “TODO” tags.	Code smell	Info
S1659	Multiple variables should not be declared on the same line.	Code smell	Minor
S1199	Nested code blocks should not be used.	Code smell	Minor
S100	Method names should comply with a naming.	Code smell	Minor

VI. Threats to Validity

In this section, we introduce the threats to validity, following the structure suggested by Yin [20] and debating the different tactics adopted to mitigate them.

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 may not be perfect, but we tried to replicate the same conditions adopted by practitioners when using the same tool. Developers prioritized the removal of the TD issues independently, without applying any specific prioritization model [21].

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 SonarQube.

External Validity. 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.

Reliability. We used standard Python packages to perform all statistical analyses since it allows simple replication of the results and gives confidence on the quality of the results.

VII. Conclusion and Future Development

In this work, we aim to understand the accuracy of the Technical Debt (TD) remediation effort that SonarQube associates to TD Items. For this purpose, we designed and conducted a case study where we asked 65 novice developers, represented by third-year undergraduate students (in Computer Science), organized in teams, to improve the quality of 15 open source projects, by choosing and fixing TD items among those identified in the SonarQube quality reports. The developers tracked the time they spent to fix each TD item. Then, we compared the actual remediation time with the one suggested by SonarQube.

The results show that, in the TD items fixed by our participants, the remediation time proposed by SonarQube tends to be overestimated. Moreover, the most accurate estimation relates to code smells, while the least accurate concerns bugs.

With this case study, we contribute to the body of knowledge by providing a first empirical study evaluating TD Remediation time. Based on the results, and taking into consideration the threats to validity, we conclude that the TD time estimated by SonarQube is overestimated, and that more empirical studies with a wider user-base will be needed to validate our results. From a practitioner's viewpoint, the results of our work are promising as they represent an initial attempt to validate the TD estimation.

Future work includes the replication of this work in industrial settings [22], [23] considering a wider range of projects where SonarQube is integrated into the development process [24]. Moreover, we are also investigating the actual impact of TD issues on the different TD areas proposed by SonarQube (i.e., maintainability, vulnerability, and security).

References

[1] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *6th International Conference in Software Engineering for Defence Applications*, 2020.

[2] V. Lenarduzzi, A. Sillitti, and D. Taibi., "Analyzing forty years of software maintenance models," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 146–148. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.122

[3] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Workshop on Managing Technical Debt*, ser. MTD '11, 2011, pp. 1–8.

[4] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," ser. Advances in Computers. Elsevier, 2011, vol. 82, pp. 25 – 46.

[5] Y. Guo, R. Spínola, and C. Seaman, "Exploring the costs of technical debt management – a case study," *Empirical Software Engineering*, vol. 21, no. 1, pp. 159–182, 2016.

[6] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Workshop on Managing Technical Debt*, 2011, pp. 31–34.

[7] R. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012, pp. 91–100.

[8] N. Saarimäki, V. Lenarduzzi, and D. Taibi, "On the diffuseness of code technical debt in open source projects of the apache ecosystem," *International Conference on Technical Debt (TechDebt 2019)*, 2019.

[9] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, 2009.

[10] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," *Encyclopedia of Software Engineering*, 1994.

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[12] M. Baldassarre, M. Piattini, F. Pino, and G. Visaggio, "Comparing iso/iec 12207 and cmmi-dev: Towards a mapping of iso/iec 15504-7," 2009, pp. 59–64.

[13] C. Pardo, F. Pino, F. García, M. Piattini, and M. Baldassarre, "A process for driving the harmonization of models," 2010, pp. 51–54.

[14] M. Baldassarre, D. Caivano, and G. Visaggio, "Software renewal projects estimation using dynamic calibration," 2003, pp. 105–115.

[15] S. Conte, H. Dunsmore, and V. Shen, "Software effort estimation and productivity," ser. Advances in Computers, 1985, vol. 24, pp. 1 – 60.

[16] M. Jorgensen, "Experience with the accuracy of software maintenance task effort prediction models," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 674–681, 1995.

[17] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit, "A simulation study of the model evaluation criterion mmre," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 985–995, 2003.

[18] I. Myrtveit, E. Stensrud, and M. Shepperd, "Reliability and validity in comparative studies of software prediction models," *IEEE Transactions on Software Engineering*, vol. 31, no. 5, pp. 380–391, 2005.

[19] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Information and Software Technology*, vol. 54, no. 8, pp. 820 – 827, 2012, voice of the Editorial Board.

[20] R. Yin, *Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5)*, 4th ed. SAGE Publications, Inc, 2009.

[21] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana, "Technical debt prioritization: State of the art. a systematic literature review," 2019.

[22] M. Baldassarre, D. Caivano, and G. Visaggio, "Empirical studies for innovation dissemination: Ten years of experience," 2013, pp. 144–152.

[23] P. Ardimento, D. Caivano, M. Cimitile, and G. Visaggio, "Empirical investigation of the efficacy and efficiency of tools for transferring software engineering knowledge," *Journal of Information and Knowledge Management*, vol. 7, no. 3, pp. 197–207, 2008.

[24] V. Lanarduzzi, A. C. Stan, D. Taibi, and G. Venters, "A dynamical quality model to continuously monitor software maintenance," in *11th European Conference on Information Systems*, 2017.