On the Diffuseness of Technical Debt Items and Accuracy of Remediation Time When Using SonarQube

Maria Teresa Baldassarre[a], Valentina Lenarduzzi[b], Simone Romano[a], Nyyti Saarimäki[c]

[a] *University of Bari, Italy*
[b] *LUT University, Finland*
[c] *Tampere University, Finland*

Abstract

Context. Among the static analysis tools available, SonarQube is one of the most used. SonarQube detects Technical Debt (TD) items—i.e., violations of coding rules—and then estimates TD as the time needed to remedy TD items. However, practitioners are still skeptical about the accuracy of remediation time estimated by the tool.
Objective. In this paper, we analyze both diffuseness of TD items and accuracy of remediation time, estimated by SonarQube, to x TD items on a set of 21 open-source Java projects.
Method. We designed and conducted a case study where we asked 81 junior developers to x TD items and reduce the TD of 21 projects.
Results. We observed that TD items are diffused in the analyzed projects and most items are code smells. Moreover, the results point out that the remediation time estimated by SonarQube is inaccurate and, as compared to the actual time spent to x TD items, is in most cases overestimated.
Conclusions. The results of our study are promising for practitioners and researchers. The former can make more aware decisions during project execution and resource management, the latter can use this study as a starting point for improving TD estimation models.

Keywords: Technical debt, remediation time, effort estimation,

Email addresses: mariateresa.baldassarre@uniba.it (Maria Teresa Baldassarre), valentina.lenarduzzi@lut.fi (Valentina Lenarduzzi), simone.romano@uniba.it (Simone Romano), nyyti.saarimaki@tuni.fi (Nyyti Saarimäki)

Published in: Information and Software Technology.
This version of the article has been accepted for publication, after peer review and is subject to Elsevier terms of use.
The Version of Record is available online at: https://doi.org/10.1016/j.infsof.2020.106377

1. Introduction

Improving software quality requires a lot of effort, and software companies have been investing in refactoring activities to remove everything that can impact the quality of their products, including technical issues [1, 2] like non-compliance with specific coding rules or with documentation conventions. Neglecting such issues can reduce the overall quality and consequently increase the *Technical Debt* (*TD*) of the entire system over time.

TD has been defined as "*making technical compromises that are expedient in the short term, but that create a technical context that increases complexity and cost in the long term*" [3]. Software companies usually adopt static analysis tools to measure software quality and TD [4, 5, 6]. Among the static analysis tools available, SonarQube[1] is one of the most used—e.g., it has been adopted by more than 100K organizations including nearly 15K public open-source projects [7]. SonarQube checks for code compliance against a set of coding rules (i.e., technical issues). If the code violates any of the classified rules, SonarQube considers it a violation or a *TD item*. Moreover, it defines TD as the time needed (i.e., *remediation time*) to refactor the violated rules (i.e., TD items).

The diffuseness of TD items in software systems (i.e., to what extent TD items are present in software systems) has been investigated in previous work [8, 9] whereas the overarching impact of TD items on software quality needs further attention [9, 10, 11]. Moreover, a recent study [12] proposed a "surgically-precise" TD estimation approach to enable a more precise and fine-grained lens of analysis over individual TD items. The results highlighted the need to keep track of the actual remediation time to fix TD items, in order to assess the accuracy of the estimated remediation time in TD tools.

In our previous work [10], we investigated the accuracy of the remediation time suggested by SonarQube to fix TD items and the diffuseness of TD items. To assess the accuracy of SonarQube's remediation time we needed to compare the actual time with the estimated one. As so, we conducted a case study where we asked 65 junior developers to remove TD items from 15 open-source Java projects. We then compared the effort (i.e., time) developers

[1] `https://www.sonarqube.org`

spent to remedy TD items against the estimation proposed by SonarQube.

This paper extends the previous one [10] as follows:

- We increased the number of participants (from 65 to 81) and number of open-source Java projects (from 15 to 21) to strengthen our results (i.e., as compared to our previous paper, we took into account 16 new participants who removed TD items on six new open-source Java projects).
- We deepened our analysis on the accuracy of TD remediation time by considering a set of coding rules, i.e., the most fixed ones (see $\mathbf{RQ}_{2.5}$ in Section 4) to have a more complete picture of the accuracy.
- We extended the analysis to all SonarQube's severity levels (see $\mathbf{RQ}_{2.2}$ in Section 4) when studying the accuracy of TD remediation time since the previous paper only considered a subset of severity levels.
- We studied the accuracy of TD remediation time by considering different effort levels (see $\mathbf{RQ}_{2.4}$ in Section 4). We also considered these effort levels when studying the diffuseness of TD items.
- We applied statistical hypothesis tests to strengthen our conclusions—in the previous paper, no statistical hypothesis test was applied.

Paper Structure. Section 2 outlines background information to better understand our research. Section 3 summarizes the existing research related to our study. In Section 4, we present the plan and execution of our case study. We report the obtained results in Section 5 and further discuss them in Section 6. Section 7 focuses on limitations and how we addressed threats to the validity of our study. In Section 8, we draw conclusions and outline future directions.

2. Background

In this section, we provide background information on TD and then on SonarQube.

2.1. Technical Debt

The increase of TD in a software product, in the long run, slows down the development process [13, 14]. Li et al. [15] conducted a systematic mapping study to understand the concept of TD and create an overview of the current state of the art regarding the management of TD. The authors proposed a classification made up of ten TD types, derived from a set of 96 selected studies. Their classification included: *requirement TD*, *architectural TD*, *design TD*, *code TD*, *test TD*, *build TD*, *documentation TD*, *infrastructure TD*, *versioning TD*, and *Defect TD*. According to the definition by Li et al. [15], code TD is related to *poorly written code*, avoiding all the *best coding practices* or *coding rules*. That said, code TD can be estimated with static analysis tools such as SonarQube, detailed in the next section.

2.2. SonarQube

SonarQube calculates several metrics such as lines of code or code complexity, and verifies the code's compliance against a specific set of coding rules defined for most common development languages. If the analyzed source code violates a coding rule, SonarQube generates an issue (i.e., a TD item). TD items are classified according to three quality characteristics, namely: *reliability*, *security*, and *maintainability*. TD items used to measure the reliability quality characteristic are called *bugs*. Those used to measure security and maintainability, are called *vulnerabilities* and *code smells*, respectively. In other words, there are three types of TD items (or coding rule), namely: bug, vulnerability, and code smell. SonarQube also assigns a *severity level* to each TD item (or coding rule), namely: *info*, *minor*, *major*, *critical*, and *blocker*.

SonarQube rates each quality characteristic according to its *quality gate*—i.e., a set of conditions based on measure thresholds against which the project is measured. In particular, each quality characteristic is rated from A to D, where A is the best value that a software project can achieve. Although SonarQube's quality gate can be customized, in our study, we used the built-in quality gate (i.e., the so-called *sonar way*) because practitioners are reluctant to customize it and usually just rely on the standard set of rules [16].

Given a TD item, SonarQube provides a TD estimation, which is defined as the time to remedy that TD item. In other words, SonarQube assumes that the TD for an item is equal to the remediation time for that item. When estimating the remediation time for a TD item, SonarQube does not specify which are the assumptions, if any, about the developer who should remedy that item—e.g., SonarQube does not specify whether, or not, it assumes that

the developer has a certain level of seniority or familiarity with the source code affected by the TD item. The estimated remediation time includes the time to comprehend the code, if any (e.g., to remove an unused import, there is no need to comprehend the code), and the time to modify the code so as to remove the TD item.

3. Related Work

In this section, we summarize the research on both diffuseness of TD items, including studies on the diffuseness of code smells, and TD.

Besides the study we extend in this paper [10], to the best of our knowledge few studies have investigated the diffuseness of TD items (i.e., bugs, vulnerabilities, and code smells) [8, 9]. In particular, Saarimäki et al. [9] investigated the diffuseness of TD items in 33 Java projects belonging to the Apache ecosystem. They found that: *(i)* TD items are diffused; *(ii)* code smells are much more diffused than bugs and vulnerabilities; and *(iii)* only a small proportion of introduced TD items is info or blocker, while most TD items are minor or major. Digkas et al. [8] studied the diffuseness of TD items by focusing on which coding rules are more violated in the Apache ecosystem. They found that the top 10 violated coding rules account for more than 40% of the coding rules violated in the Apache ecosystem. The above-mentioned studies both investigate TD items detected through SonarQube but, unlike our study, they focus on projects from the Apache ecosystem only.

Conversely, there are several studies that have focused on the diffuseness and/or evolution of code smells [17, 18, 19, 20, 21] from Fowlers' catalog [22] and/or Brown et al.'s catalog [23] (i.e., they do not focus on SonarQube's code smells). These studies show that: *(i)* some code smells are diffused while others are not [19]; *(ii)* the presence of code smells increases over time [17, 20]; and *(iii)* code smells survive long [18, 21]. Moreover, most code smells are introduced when an artifact (e.g., a class) is created; while, in almost 400 cases, code smells are introduced by refactoring activities [21].

Most of the research on TD aims to define approaches that quantify TD in terms of cost to fix technical issues [24] or that conceptualize the relationship between cost and benefit to improve software quality and help the decision-making process during maintenance activities [25]. Another model estimates (defect) TD based on the concept that maintenance costs increase over time due to code degradation [26]. Zazworka et al. [27] focused on automated identification of TD comparing it with manual identification provided by

developers. The results showed a small overlap between the two estimations and pointed out how the adoption of tools supports defect identification.

Some studies have investigated SonarQube's TD items by considering change- and fault-proneness [7, 11, 28, 29]. Furthermore, existing research highlights that developers are not completely sure about the usefulness of the rules [16, 30] provided by SonarQube. Moreover, developers refactor their code according to high severity levels of the identified violations to reduce the risk of faults [16, 30]. These concerns are also confirmed in another study [7] that examined the fault-proneness of SonarQube TD items, in order to identify which are actually fault-prone and to assess the fault-prediction model accuracy. Analyzing the different types and severity levels of SonarQube's TD items, no significant difference between the clean and infected classes was found [11]. Furthermore, considering the three different types (i.e., bug, vulnerability, and code smell), the results showed a small effect on change-proneness and no effect on fault-proneness [11].

The estimation of SonarQube's TD 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 TD does not seem to have a large impact on the lead time to add functionality and fix bugs [12].

Despite the studies mentioned in this section are related to TD, none of them focuses of the accuracy of TD remediation time that SonarQube suggests.

4. Case Study Design

We designed our case study by taking into account the recommendations by Runeson and Höst [31]. To allow replicating the case study, we share both raw data and analysis scripts by means of our replication package.[2]

4.1. Goal and Research Questions

The goal of the case study can be formalized, by using the Goal Question Metric (GQM) template [32], as follows:

> ***Analyze*** *TD items that SonarQube identifies* ***for the purpose of*** *assessing them* ***with respect to*** *their diffuseness and accuracy*

[2] https://doi.org/10.6084/m9.figshare.11673132

of estimated remediation time to fix them ***from the point of view of*** *both practitioner and researcher* ***in the context of*** *open-source Java projects and junior developers.*

Based on the above-mentioned goal, we formulated and then investigated the following Research Questions (RQs):

RQ$_1$ What is the diffuseness of introduced TD items?

RQ$_{2.1}$ What is the accuracy of TD remediation time?

RQ$_{2.2}$ What is the accuracy of TD remediation time with respect to different severity levels?

RQ$_{2.3}$ What is the accuracy of TD remediation time with respect to different types?

RQ$_{2.4}$ What is the accuracy of TD remediation time with respect to different effort levels?

RQ$_{2.5}$ What is the accuracy of TD remediation time with respect to different coding rules (in particular, by considering the 20 coding rules that are fixed the most)?

With **RQ**$_1$ we aim to determine the prevalence of TD items that developers introduce in open-source projects written in Java. To answer **RQ**$_1$, we examine the prevalence of TD items in general, as well as per severity level, type, effort level, and coding rule. The results regarding this RQ allows understanding to what extent TD items are present in software projects. If the magnitude of the phenomenon is small—i.e., TD items are rarely present in software systems—then studying the accuracy of the remediation time of TD items might not be worthwhile.

As for the remaining RQs (i.e., from **RQ**$_{2.1}$ to **RQ**$_{2.5}$), we aim to assess the accuracy of TD remediation time that SonarQube estimates by contrasting actual remediation time against estimated one. To that end, we focus on the accuracy of TD items in general, as well as with respect to their severity level, type, effort level, and coding rule.

While the severity level, type, and coding rule corresponding to each TD item are explicitly returned by SonarQube, the effort level is not. By bearing

in mind the effort levels mentioned in SonarQube's documentation[3] (i.e., trivial, easy, medium, sizeable,[4] high, and complex), we classified each TD item, based on the estimated remediation time, as: *trivial*, if the remediation time is in the interval $(0, 10]$[5] minutes; *easy* if the remediation time is in the interval $(10, 20]$ minutes; *medium* if the remediation time is in the interval $(20, 30]$ minutes; *sizeable* if the remediation time is in the interval $(30, 60]$ minutes; *high* if the remediation time is in the interval $(60, 180]$ minutes; and *complex*, otherwise.

4.2. Context

We focused the case study on open-source projects written in Java, which satisfied the following four criteria. First, the projects had to be hosted on GitHub[6] and, second, their size had to be greater than 10KLOC (i.e., 10,000 Lines Of Code). These two criteria were imposed in order to select open-source projects with a large enough size. Third, the analysis of SonarQube on those projects had to reveal that at least two SonarQube's ratings out of three (i.e., for bugs, vulnerabilities, and code smells) were less than A. This was to avoid the inclusion of projects with few TD items and thus force the participants in our study to spend a significant amount of time remediating TD items. Nevertheless, we are aware that, in some projects, some developers could use SonarQube to remediate TD items so leading those projects to have fewer TD items than they would have in absence of SonarQube (see Section 7 where we discuss such a validity threat). Finally, to provide the participants with a safety net when remediating TD items, the projects had to have a regression test suite.

In Table 1, we summarize some information on the projects considered in our study. By looking at this table, we can grasp the extent to which the projects are heterogeneous in terms of estimated remediation time, lines of code, number of classes (i.e., #Classes), McCabe's Cyclomatic Complexity (i.e., CC) [33], and number of tests (i.e., #Tests). We can also notice that, although the estimated remediation time for the code smell type is larger

[3] docs.sonarqube.org/8.1/extend/adding-coding-rules

[4] SonarQube's documentation uses the term major, instead of the term sizeable. We opted for the latter term to avoid confusion between the major effort level and the major severity level.

[5] The mathematical notation $(a, b]$ indicates that the interval includes b but not a.

[6] github.com

than that for the bugs and vulnerabilities types, the rating of SonarQube is always equal to A.

The case study was executed within the Computer Science degree at the University of Bari (Bari, Italy). The participants were last-year undergraduate students who were taking the *Software Quality* course. This course included both face-to-face and laboratory classes, which covered the following topics: software quality (i.e., internal, external, and in-use); ISO standards for software quality; software quality assessment, monitoring, and improvement [34, 35]; supporting tools for quality management (e.g., SonarQube); and process control [36, 37]. The students had to carry out an assignment by working in teams, which consisted of improving the internal quality of a Java project from the reliability, security, and maintainability perspectives. To that end, the participants had to remedy TD items related to these perspectives—i.e., they had to fix bugs, vulnerabilities, and code smells, respectively.

According to some demographic information we gathered through a prequestionnaire at the beginning of the Software Quality course, the background of the students was quite homogeneous. In particular, all but one student, who took part in the study, had passed the exam about object-oriented programming (with Java), with high marks. Moreover, their self-reported experience in Java, on a 5-point scale from "very inexperienced" (1) to "very experienced" (5), was median of 3.5. Furthermore, the Computer Science degree curriculum includes two capstone projects, each consisting of tight collaboration between students and industry partners. More precisely, the industry partners, usually local ICT companies, provide a set of requirements concerning a challenging topic/social problem that students, organized in teams, provide a solution to. At the end of the course, the students presented their solution (e.g., an app or a web app) to a panel of industry experts (customers), community members (end users), and professors of the course, who collectively evaluate the quality of their solution. To develop their solution, the students followed Scrum and used an application life-cycle management tool to manage both team and project. The question of whether students can be used as subjects in software engineering studies is widely debated (e.g., [38], [39]). Our study involved last-year undergraduate students knowledgeable on the concepts of TD with a good experience in software development. We feel that the sample of 81 students who participated in the study are representative of junior developers since their participation in two capstone projects involved heavy and direct collab-

Table 1: 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
docker-maven-plugin (github.com/fabric8io/docker-maven-plugin)	3h 30m (E)	1h 25m (E)	6d 6h (A)	16	250	3,481	442
Flickr4Java (github.com/boncey/Flickr4Java)	1h 30m (E)	6h 45m (B)	15d (A)	14	166	2,957	135
FXGL (almasb.github.io/FXGL)	2h 34m (D)	6d 4h (B)	24d (A)	23	251	4,503	301
GameComposer (github.com/mirkosertic/GameComposer)	3h 52m (E)	2d (B)	59d (A)	22	469	5,451	233
getting-started-java (github.com/GoogleCloudPlatform/getting-started-java)	1d (E)	11d (B)	14d (A)	11	185	1,236	36
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
JFreeChart (github.com/jfree/jfreechart)	7d 1h (D)	6h 55m (B)	77d (A)	94	652	18,793	2,176
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
MovSim (github.com/movsim/movsim)	2h 32m (D)	2h 50m (B)	10d (A)	17	250	3,480	61
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
VeraPDF (github.com/veraPDF/veraPDF-library)	1h 50m (E)	1h 20m (D)	10d (A)	26k	363	5,289	262

* Information gathered by using SonarQube.
° Information gathered by using Apache Maven (maven.apache.org).
† d, h, and m stand for days, hours, and minutes, respectively.

oration with industry partners who outlined the requirements and interacted with them during the project in the role of customers. At the time of our research, all of the students involved had carried out both capstone projects and some students had even been contacted by the companies for their internships given the positive impression they had made. Furthermore, the students were close to graduation and it is likely they would have worked within a range of six months.

4.3. Procedure and Data Collection

To select the Java projects to include in the case study and then collect the actual time it takes to remedy TD items, we leveraged the assignments we gave to the students of the Software Quality course. In particular, we defined and then put into practice the following protocol:

1. The students were asked to form mutually-exclusive teams. Each team consisted of two to five participants, including a team leader appointed by the members of the team.

2. Each team had to search for an open-source project written in Java satisfying the criteria mentioned in Section 4.2. When a team identified a project satisfying those criteria, the team leader would send a project proposal to the authors MTB and SR, who approved or denied it. A proposal had to include a SonarQube report to let MTB and SR *(i)* evaluate whether the criteria were satisfied or not and *(ii)* establish target values of TD that any team had to achieve. In particular, every team had to achieve a triple A for the reliability, security, and maintainability perspectives and, based on both characteristics of the proposed project (e.g., overall estimated remediation time) and team size, had to achieve a TD of at most 2-3 days.

3. Once the project was approved, the team had to define an *action plan*. That is, the team members autonomously chose, among the TD items SonarQube identified, which ones they wanted to fix in order to achieve the target values of TD established for that project. To define the action plan, the team was supported by Redmine,[7] an application lifecycle management tool. For each TD item included in the action plan,

[7] www.redmine.org

Redmine stored information like: identification number of the TD item; status (at the beginning, the status of each TD was set to *open*, then it could become *in-progress*, *fixed*, etc.); type (i.e., bug, vulnerability, or code smell); description of the TD item including the corresponding *squid*—it is an identifier for coding rules (e.g., *S1141* is the squid identifying the rule *"Try-catch blocks should not be nested"*)—; time estimated by SonarQube to fix the TD item; and assignee (i.e., the team member who had to fix the TD item). Through Redmine, the team also developed a Gantt chart depicting the assignment of the team members to the activities (i.e., the fixing of TD items) over time.

4. When fixing TD items, the team members were supported by Redmine. Thanks to this tool the team members kept track, by using appropriate tags, of the actual time it took to remedy TD items. In particular, when a team member deemed to have fixed a given TD item, he/she committed the changes into the version control system by specifying, in the commit message, a tag that allowed linking the remediation time, spent by the team member, with the TD item. For example, the tag "refs #12 @1h30m" allowed Redmine to automatically store the actual remediation time (i.e., 1 hour and 30 minutes) of the TD item #12. A TD item could also be fixed through multiple commits. In this case, each commit message had to report the tag to link the spent time with the TD item. For example, a new commit reporting the tag "refs #12 @10m", after the commit with the tag "refs #12 @1h30m", allowed Redmine to add 10 minutes to the actual remediation time stored for the TD item #12, which became equal to 1 hour and 40 minutes. It is worth mentioning the actual remediation time included both time spent comprehending the code affected by the TD item (since it is considered by the remediation time that SonarQube estimates) and time spent modifying the code to remedy the TD item. On the other hand, the actual remediation time does not include the time to run the test suite nor the time to run a new analysis with SonarQube. Once a TD item was fixed, the assignee of that TD item could change its status to fixed.

5. Once the team achieved the established target values of TD, we could extract, for each fixed TD item, information like squid, estimated and actual remediation time, or type (i.e., bug, vulnerability, or code smell). Other information could be derived like severity level (i.e., info, minor,

major, critical, or blocker) and effort level (i.e., trivial, easy, medium, sizeable, high, or complex).

4.4. Data Analysis

To study $\mathbf{RQ}_1$, we counted the TD items that SonarQube identified in the selected projects.

As for the the other RQs (i.e., from $\mathbf{RQ}_{2.1}$ to $\mathbf{RQ}_{2.5}$), we used the following metrics to evaluate the accuracy of SonarQube's estimated remediation time, namely:

Mean(RE). It is the mean of Relative Errors (REs). More formally, this metric is defined as [40]:

$$Mean(RE) = \frac{1}{n}\sum_{i=1}^{n} RE_i \tag{1}$$

where n indicates, in our case, the number of SonarQube's estimations (one SonarQube's estimation per fixed TD item), while RE_i is the RE of the i-th SonarQube's estimation (i.e., the estimation of the i-th fixed TD item), namely:

$$RE_i = \frac{actual\_remediation\_time_i - estimated\_remediation\_time_i}{actual\_remediation\_time_i} \tag{2}$$

According to Conte et al. [40], good estimations are characterized by a small value of $Mean(RE)$. 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 [40]. Positive values of $Mean(RE)$ indicate that, on average, the remediation time that SonarQube suggests is underestimated. On the other hand, negative values of $Mean(RE)$ indicates that, on average, such remediation time is overestimated. For example, a $Mean(RE)$ value of 0.3 means that, on average, the remediation time is underestimated by 30%. On the other hand, a $Mean(RE)$ value of -0.3 means that, on average, the remediation time is overestimated by 30%.

MMRE. The Mean Magnitude of RE ($MMRE$) metric is defined as [40]:

$$MMRE = \frac{1}{n}\sum_{i=1}^{n} MRE_i \tag{3}$$

where n indicates the number of SonarQube's estimations, while MRE_i is the magnitude of the RE of the i-th SonarQube's estimation, namely:

$$MRE_i = \frac{|actual\_remediation\_time_i - estimated\_remediation\_time_i|}{|actual\_remediation\_time_i|} \quad (4)$$

In line with the general interpretation of $MMRE$ [40], if the value of $MMRE$ is small, then SonarQube should produce, on average, good estimations. It is worth noting that $MMRE$ may be influenced by few very high values of MRE_i [41]. From a practical point of view, an $MMRE$ value of 0.3 means that, on average, the magnitude of RE of an estimation is 30%.

MdMRE. The Median Magnitude of RE ($MdMRE$) metric has been proposed to limit the impact of few very high values of MRE_i [41]. It is easy to grasp that, given n SonarQube's estimations, $MdMRE$ is computed as the median of the MREs of these estimations [41]. The lower the $MdMRE$ value is, the better SonarQube's estimations are. From a practical point of view, an $MdMRE$ value of 0.3 means that, in the median case, the magnitude of RE of an estimation is 30%.

PRED25. This metric is defined as the percentage of estimations where $MRE_i \leq 0.25$ [41]. In other words, $PRED25$ measures the amount of TD remediation time estimated by SonarQube that falls within 25% of the actual remediation time. The higher the $PRED25$ value is, the better SonarQube's estimations are.

PRED50. Similar to $PRED25$, this metric is defined as the percentage of estimations where $MRE_i \leq 0.50$ [41]. It is easy to grasp that $PRED50$ measures the amount of TD estimated remediation time that falls within 50% of the actual remediation time. Again, the higher the $PRED50$ value is, the better SonarQube's estimations are.

SdMRE. To have a measure of variability of the magnitude of the REs, we used the Standard deviation of Magnitude of RE ($SdMRE$) metric. It is easy to grasp that, given n SonarQube's estimations, $SdMRE$ is computed as the standard deviation of the MREs of these estimations. The lower the $SdMRE$ value is, the better it is.

We computed the above-mentioned metrics for each project and by considering all projects together. According to Jørgensen et al. [41], the accuracy metrics $MMRE$, $MdMRE$, $PRED25$, and $PRED50$ measure different properties of a prediction model. The same holds for $Mean(RE)$, which

allows understanding whether a prediction model tends to produce overestimations or underestimations, and $SdMRE$, which is a measure of variability. Therefore, the use of six metrics to evaluate the accuracy of estimated remediation time should allow us to have a more complete and correct picture of accuracy so mitigating a threat of *mono-method bias* [42].

5. Results

In this section, we report the results with respect to each RQ.

5.1. RQ_1 — What is the diffuseness of introduced TD items?

In Table 2, we summarize the diffuseness of TD items the original developers of the analyzed projects had introduced in those projects. As shown in Table 2, TD items were introduced, overall, 28,436 times. Moreover, the number of introduced TD items varies from a minimum of 468 (docker-maven-plugin) to a maximum of 3,365 (TrackMate). These results seem to suggest that the presence of TD items is quite common in the analyzed projects.

Table 2 also shows that, when considering all projects together, the severity of most introduced TD items is minor (11,655) or major (10,317), followed by critical ones (5,337). As for the info and blocker severity levels, a much smaller number of introduced TD items can be observed in the analyzed projects (289 and 838, respectively). This trend is confirmed when considering the projects individually (see either Table 2 or the left-hand side of Figure 1).

As for the type of introduced TD items, the results summarized in Table 2 show that, overall, code smells are much more diffuse than bugs (25,497 vs. 1,225) and vulnerabilities (25,497 vs. 1,714). This trend is confirmed when looking at the projects individually (see either Table 2 or the right-hand side of Figure 1).

Regarding the effort level, if we consider all projects together, most introduced TD items are trivial (22,453), followed by easy (4,331) and medium (1,503). In particular, trivial, easy, and medium TD items account for 99% of introduced TD items. Table 2 and Figure 1 (the bottom part) confirm the above-mentioned trend within each project.

Among the coding rules that SonarQube checks, 251 were violated in the analyzed projects. Table 3 shows the number of TD items concerning the top ten violated coding rules (see Appendix A for a description of these coding

Table 2: Diffuseness of introduced TD items for each project. The number of introduced TD items is also grouped by severity level, type, and effort level.

Project	**Severity Level**[*]					**Type**[°]			**Effort Level**[◁]						**Total**
	In	Mi	Ma	Cr	Bl	B	V	CS	Tr	Ea	Me	Si	Hi	Co	
Apache PDFBox	74	718	900	621	61	74	106	2,194	1,940	314	114	0	0	6	2,374
Computoser	4	151	285	70	5	31	33	451	421	75	18	0	0	1	515
docker-maven-plugin	16	216	130	101	5	13	7	448	379	58	31	0	0	0	468
Flickr4Java	70	934	388	206	5	14	40	1,549	1,432	111	60	0	0	0	1,603
FXGL	0	1,451	612	157	22	18	298	1,926	2,084	122	32	0	0	4	2,242
GameComposer	0	683	373	234	77	32	85	1,250	1,143	126	31	0	0	67	1,367
getting-started-java	0	403	309	95	50	27	327	503	450	394	13	0	0	0	857
IRI	0	252	242	82	12	22	54	512	424	119	45	0	0	0	588
jChecs	0	227	178	383	22	54	10	746	494	302	12	0	0	2	810
JFreeChart	2	1,072	1,127	536	214	287	60	2,604	2,064	466	412	0	0	9	2,951
jsoniter	0	262	210	121	1	20	84	490	443	61	90	0	0	0	594
Libresonic	2	281	628	218	26	58	31	1,066	676	406	73	0	0	0	1,155
MovSim	0	172	359	93	2	20	22	584	558	31	33	0	0	4	626
MyBatis	20	466	857	385	16	159	32	1,553	1,163	415	165	1	0	0	1,744
Ninja	61	466	440	153	15	19	81	1,035	846	213	75	0	0	1	1,135
OkHttp	6	359	569	214	48	69	36	1091	993	179	22	2	0	0	1,196
OpenAudible	0	652	729	397	0	42	139	1,597	1,476	269	28	0	0	5	1,778
RoaringBitmap	34	620	601	250	84	72	60	1,457	1,246	238	105	0	0	0	1,589
Traccar	0	157	374	332	16	98	23	758	740	106	23	0	0	10	879
TrackMate	0	1,868	840	511	146	84	179	3,102	2,955	271	102	0	0	37	3,365
VeraPDF	0	245	166	178	11	12	7	581	526	55	19	0	0	0	600
All projects	289	11,655	10,317	5,337	838	1,225	1,714	25,497	22,453	4,331	1,503	3	0	146	28,436

[*] In, Mi, Ma, Cr, and Bl stand for Info, Minor, Major, Critical, and Blocker, respectively.
[°] B, V, and CS stand for Bug, Vulnerability, and Code Smells, respectively.
[◁] Tr, Ea, Me, Si, Hi, and Co stand for Trivial, Easy, Medium, Sizable, High, and Complex, respectively.

rules), which account for 39% of introduced TD items—i.e., among the 28,436 introduced TD items, 10,992 concern the top ten violated coding rules.

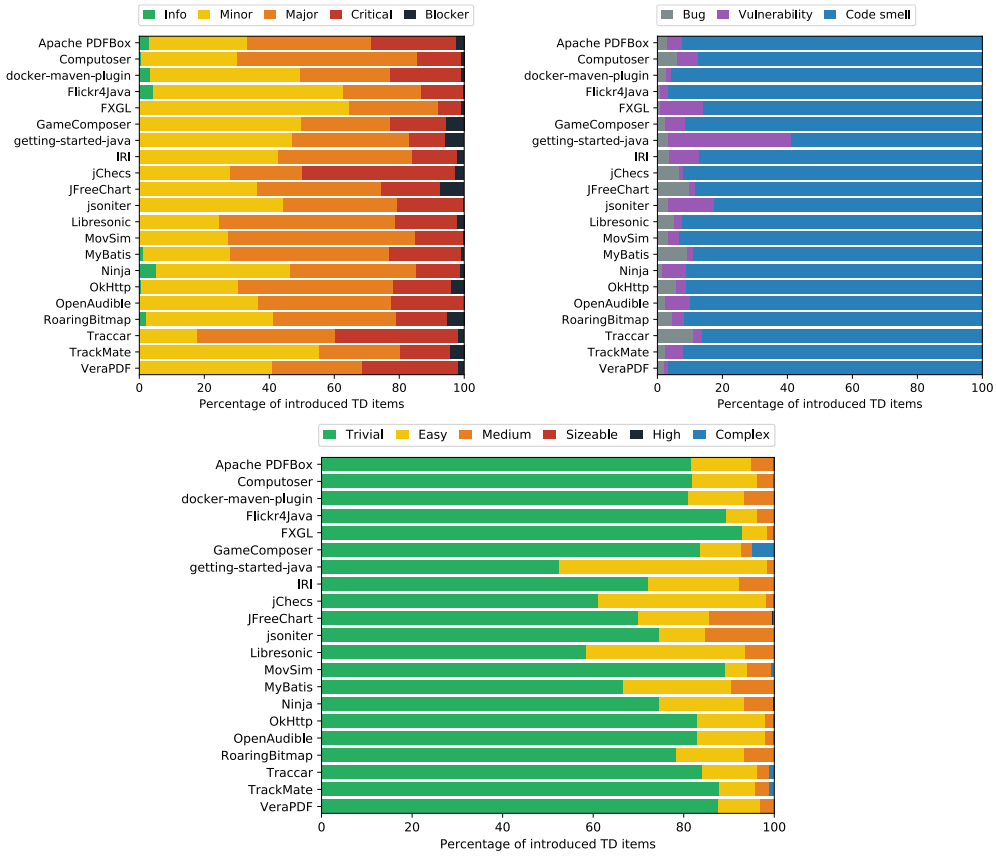

Figure 1: Percentage stacked bar charts depicting the proportions of introduced TD items per severity level (top left-hand side), type (top right-hand side), and effort level (bottom).

In Figure 2, we show the boxplots depicting the distributions of introduced TD items for those coding rules that are violated, overall, at least 300 times. We can observe that the distributions are right-skewed and the mean value is always greater than the median value. Although this indicates that, in some projects, certain coding rules are violated much more as compared to other projects, we can also observe that the median values are generally much greater than zero. That is to say that the TD items that violate these coding rules are, in general, diffused among the analyzed projects.

Table 3: Diffuseness of introduced TD items, for each project, with respect to the top ten violated coding rules, each identified by a squid.

Project	Squid										Total
	S2293	**S3776**	**S1192**	**S112**	**S125**	**db**	**S116**	**S106**	**S115**	**S1659**	
Apache PDFBox	2	248	138	48	27	92	11	250	105	50	971
Computoser	35	39	17	10	25	5	5	68	0	2	206
docker-maven-plugin	4	11	10	5	25	2	9	5	41	18	130
Flickr4Java	546	8	133	7	17	39	2	6	9	28	795
FXGL	16	58	3	9	289	20	478	3	68	178	1,122
GameComposer	3	34	31	13	39	19	22	2	32	96	291
getting-started-java	7	1	44	0	0	130	0	0	0	0	182
IRI	5	19	8	64	25	7	6	4	7	40	185
jChecs	0	88	8	0	3	19	173	10	5	0	306
JFreeChart	60	212	77	103	89	187	4	8	0	72	812
jsoniter	0	49	36	13	4	34	0	8	2	1	147
Libresonic	167	42	131	270	13	23	2	25	1	1	675
MovSim	1	24	3	0	120	8	6	27	16	0	205
MyBatis	4	37	284	308	19	180	7	12	6	1	858
Ninja	17	26	18	86	27	5	12	21	34	0	246
OkHttp	0	50	36	47	32	8	4	113	35	2	327
OpenAudible	8	34	50	51	243	10	14	58	229	35	732
RoaringBitmap	45	124	36	162	68	67	29	22	21	142	716
Traccar	1	138	177	40	4	76	0	0	0	15	451
TrackMate	727	98	65	3	71	114	36	139	93	74	1,420
VeraPDF	1	52	49	2	15	10	19	4	60	3	215
All projects	1,649	1,392	1,354	1,241	1,155	1,055	839	785	764	758	10,992

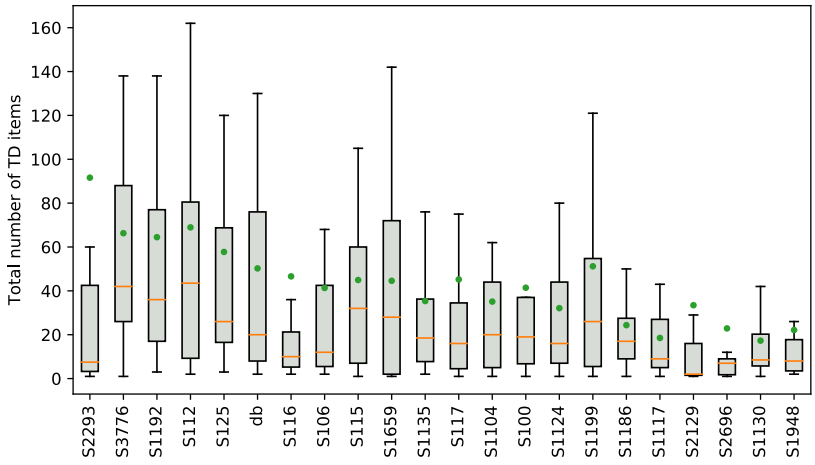

Figure 2: Boxplots depicting the distributions of introduced TD items for the coding rules that, considering all projects together, are violated at least 300 times.

5.2. $RQ_{2.1}$ — What is the accuracy of TD remediation time?

In Table 4, we show the diffuseness of TD items that the participants in our study fixed. These TD items represent the sample we used to study $\mathbf{RQ}_{2.1}$. As shown in Table 4, the number of fixed TD items ranges from a minimum of 24, for GameComposer, to a maximum of 832, for getting-started-java. In total, the participants fixed 3,636 TD items.

The results about the accuracy of TD remediation time that SonarQube

Table 4: Diffuseness of fixed TD items for each project. The number of fixed TD items is also grouped by severity level, type, and effort level.

Project	Severity Level*					Type°			Effort level◁						Total
	In	Mi	Ma	Cr	Bl	B	V	CS	Tr	Ea	Me	Si	Hi	Co	
Apache PDFBox	0	48	45	4	6	14	9	80	76	18	9	0	0	0	103
Computoser	0	10	16	12	2	11	6	23	24	14	2	0	0	0	40
docker-maven-plugin	0	4	22	11	4	3	1	37	27	10	4	0	0	0	41
Flickr4Java	1	31	26	5	1	3	5	56	47	13	4	0	0	0	64
FXGL	0	209	182	52	27	20	64	386	414	34	22	0	0	0	470
GameComposer	0	14	8	1	1	8	12	4	15	5	0	0	0	4	24
getting-started-java	0	295	392	95	50	27	295	510	386	434	12	0	0	0	832
IRI	0	23	30	10	2	6	5	54	44	16	5	0	0	0	65
jChecs	0	54	63	84	2	14	4	185	175	21	6	0	0	1	203
JFreeChart	0	10	13	4	1	11	4	13	18	7	3	0	0	0	28
jsonter	0	26	38	18	0	9	3	70	57	21	4	0	0	0	82
Libresonic	0	8	13	3	2	17	7	2	14	10	2	0	0	0	26
MovSim	0	4	327	11	2	9	1	334	316	21	7	0	0	0	344
MyBatis	1	13	94	39	3	13	5	132	79	47	8	0	0	16	150
Ninja	0	94	65	12	15	27	57	102	127	39	20	0	0	0	186
OkHttp	0	6	18	2	3	10	4	15	19	8	1	1	0	0	29
OpenAudible	0	16	23	12	1	5	6	41	34	15	3	0	0	0	52
RoaringBitmap	0	0	496	0	0	0	0	496	317	150	29	0	0	0	496
Traccar	0	15	29	6	7	6	2	49	39	12	5	0	0	1	57
TrackMate	0	29	21	4	1	8	6	41	34	15	5	0	0	1	55
VeraPDF	0	51	141	85	12	13	9	267	230	37	22	0	0	0	289
All projects	2	960	2,062	470	142	234	505	2,897	2,492	947	173	1	0	23	3,636

* In, Mi, Ma, Cr, and Bl stand for Info, Minor, Major, Critical, and Blocker, respectively.
° B, V, and CS stand for Bug, Vulnerability, and Code Smells, respectively.
◁ Tr, Ea, Me, Si, Hi, and Co stand for Trivial, Easy, Medium, Sizable, High, and Complex, respectively.

Table 5: TD remediation time accuracy for each project.

Project	Mean(RE)	MMRE	MdMRE	PRED25	PRED50	SdMRE
Apache PDFBox	0.27	0.92	0.62	25%	42%	1.4
Computoser	-0.15	0.92	0.71	15%	30%	1.67
docker-maven-plugin	0.02	0.60	0.50	28%	43%	0.67
Flickr4Java	-0.09	0.99	0.76	16%	24%	2.34
FXGL	-0.30	0.85	0.76	14%	28%	1.64
GameComposer	0.87	1.81	0.71	13%	33%	4.8
getting-started-java	-0.55	0.59	0.71	14%	23%	0.49
IRI	-0.81	0.92	1.00	8%	11%	0.32
jChecs	-0.02	0.48	0.25	48%	65%	0.78
JFreeChart	-0.48	0.49	0.50	32%	50%	0.35
jsonter	-0.18	0.98	0.72	15%	22%	2.61
Libresonic	2.72	3.30	1.00	12%	19%	5.53
MovSim	-0.64	0.68	0.75	8%	15%	0.26
MyBatis	-0.35	0.47	0.39	41%	53%	0.38
Ninja	-0.11	0.96	0.79	16%	27%	1.32
OkHttp	0.20	0.78	0.50	17%	48%	0.86
OpenAudible	-0.67	0.71	0.76	6%	14%	0.23
RoaringBitmap	-0.22	0.35	0.00	53%	63%	0.64
Traccar	0.25	0.84	0.49	37%	51%	1.55
TrackMate	-0.36	0.59	0.59	24%	36%	0.48
VeraPDF	-0.25	0.68	0.63	21%	36%	0.76
All projects	-0.30	0.68	0.67	23%	35%	1.21

estimates—i.e., the values of the accuracy metrics $Mean(RE)$, $MMRE$, $MdMRE$, $PRED25$, $PRED50$, and $SdMRE$ (see Section 4.4)—are summarized in Table 5. The values of the accuracy metrics clearly indicate that the estimated TD remediation time, for RoaringBitmap, is more accurate as compared to any other analyzed project (e.g., $MdMRE = 0.00$). With the only exception of RoaringBitmap, the results in Table 5 seem to suggest that SonarQube's estimations are inaccurate. For example, if we consider the $PRED25$ metric, most projects have $PRED25$ values less than or equal to 25%—only in one case (i.e., for RoaringBitmap) the $PRED25$ value is greater than 50%. This is to say that, in most projects, the amount of estimated TD remediation time that falls within 25% of the actual remediation time is less than or equal to 25%—only for RoaringBitmap the amount of estimated TD remediation time that falls within 25% of the actual remediation time is greater than 50%. The scatterplot in Figure 3 confirms that estimated TD remediation time is, in general, not accurate as many points are far from the ideal line.

Finally, we can notice that the TD remediation time suggested by SonarQube is, in general, overestimated. This is because, for 15 out of 21 projects, the $Mean(RE)$ values are negative—when considering all projects together, the $Mean(RE)$ value (-0.26) is negative as well. The scatterplot in Figure 3 also shows that TD remediation time suggested by SonarQube is, in general, overestimated—the points above the ideal line are many more than those below the ideal line. To further support this finding, we ran a one-tailed

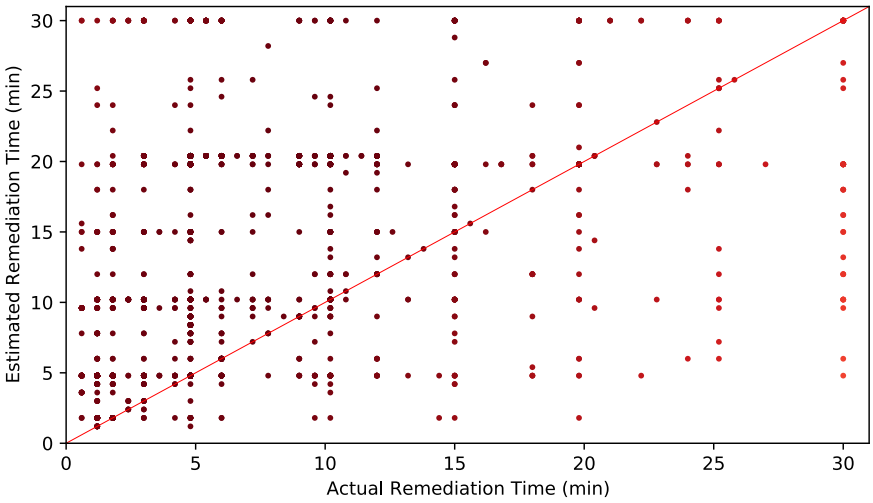

Figure 3: Scatterplot of actual remediation time vs. estimated remediation time. The darker the points are, the higher the number of overlapping points. We cut the scatterplot to the interval $[0, 30]$ minutes for both axes to improve the readability of the scatterplot.

Mann-Whitney U test[8] whose alternative hypothesis (H_1) was: *the spent remediation time is significantly less than the estimated remediation time.* As is customary in SE studies, we fixed the significance level, α, at 0.05. The p-value returned by the Mann-Whitney U test was approximately equal to 0 (i.e., less than $\alpha = 0.05$) so indicating that the spent remediation time is significantly less than the estimated one. We can, therefore, confirm that the TD remediation time suggested by SonarQube is overestimated.

5.3. $RQ_{2.2}$ — What is the accuracy of TD remediation time with respect to different severity levels?

As shown in Table 4, most fixed TD items are major (2,062). The participants also fixed a good number of minor (960), critical (470), and blocker (142) TD items. As for the info severity level, only 2 TD items were fixed.

[8] We could not apply the t-test, which is the parametric alternative to the Mann-Whitney U test [42], because the normality assumption behind that test was not met (as the Shapiro-Wilk test suggested). From here onwards, if we use the Mann-Whitney U test, instead of the t-test, it is because the normality assumption is not met.

Due to the scant number of fixed TD items classified as info, we excluded the info severity level from the analyses and conclusions that follow.

In Table 6, we show the results on the accuracy of TD remediation time per severity level. By looking at this table, it seems that SonarQube's estimations are inaccurate regardless of the severity level. For example, we can observe that most $PRED25$ and $PRED50$ values are less than or equal to 25% and 50%, respectively, for any severity level. The same conclusion can be drawn if we consider the $MMRE$ values (as well as the $MdMRE$ ones) since most values are greater than 0.25 for any severity level—i.e., for most SonarQube's estimations, the RE is, on average, greater than 25% whatever the severity level is.

As for $Mean(RE)$, it seems that the remediation time suggested by SonarQube tends to be overestimated for any severity level. It is worth mentioning that, although the overall $Mean(RE)$ value is positive (0.15) for the Blocker severity level, this value is strongly influenced by two projects (i.e., Libresonic and GameComposer). That is to say that there seems to be an overestimation of the remediation time for the Blocker severity level as well. This is because the $Mean(RE)$ values are negative for 11 projects (out of 19). We used the one-tailed Mann-Whitney U test to confirm that, within each severity level, *the spent remediation time was significantly less than the estimated remediation time* (H_1). The Mann-Whitney U test returned a p-value approximately equal to zero for any severity level so confirming that, whatever the severity level is, the remediation time suggested by SonarQube is overestimated.

With the only exception of the blocker severity level, no substantial difference in the accuracy of estimated remediation time seems to emerge among the severity levels. Indeed, we can notice that, for the blocker severity level, the amount of $PRED25$ values equal to 0% is greater as compared to any other level. To better determine if there were (significant) differences in the values of each accuracy metric due to the severity levels, we used the Kruskal-Wallis test[9]—H_1 was: *there is a significant difference in the distributions of values, for the considered accuracy metric, among the severity*

[9]We checked the assumptions of normality and homoscedasticity to apply the one-way ANOVA test, which is the parametric alternative to the Kruskal-Wallis test [42]. To do so, we used the Shapiro-Wilk test and Levene's test, respectively. In no case, the assumptions of normality and homoscedasticity were both met. From here onwards, if we use the Kruskal-Wallis test, instead of the one-way ANOVA test, it is because the assumptions of normality and homoscedasticity are not met.

Table 6: TD remediation time accuracy for each project grouped by severity level.

Severity Level	Project	Mean(RE)	MMRE	MdMRE	PRED25	PRED50	SdMRE
Info	Flickr4Java	-0.97	0.97	0.97	0%	0%	-
	MyBatis	-0.53	0.53	0.53	0%	0%	-
	All projects (2/21)	-0.75	0.75	0.75	0%	0%	0.31
Minor	Apache PDFBox	0.27	0.89	0.63	21%	42%	1.21
	Computoser	-0.61	0.61	0.67	10%	30%	0.29
	docker-maven-plugin	-0.64	0.64	0.59	0%	0%	0.15
	Flickr4Java	-0.36	0.51	0.58	32%	39%	0.36
	FXGL	-0.34	0.86	0.80	10%	28%	1.37
	GameComposer	-0.47	0.58	0.63	14%	36%	0.27
	getting-started-java	-0.67	0.67	0.74	2%	9%	0.14
	IRI	-0.66	0.88	1.00	17%	17%	0.48
	jChecs	0.08	0.27	0.00	77%	83%	0.99
	JFreeChart	-0.38	0.38	0.39	40%	60%	0.34
	jsonter	-0.44	0.71	0.69	15%	23%	0.41
	Libresonic	1.76	2.01	1.00	0%	13%	1.89
	MovSim	-0.99	0.99	1.00	0%	0%	0.03
	MyBatis	0.13	0.50	0.50	46%	46%	0.55
	Ninja	-0.06	0.95	0.74	18%	29%	1.4
	OkHttp	0.97	1.49	1.20	0%	33%	1.24
	OpenAudible	-0.74	0.74	0.73	0%	6%	0.15
	Traccar	0.82	1.12	0.25	47%	60%	2.1
	TrackMate	-0.20	0.61	0.50	28%	38%	0.61
	VeraPDF	-0.43	0.66	0.57	22%	37%	0.5
	All projects (20/21)	-0.34	0.74	0.71	16%	27%	0.96
Major	Apache PDFBox	0.08	0.82	0.52	33%	47%	1.63
	Computoser	-0.68	0.73	0.75	0%	13%	0.18
	docker-maven-plugin	0.16	0.68	0.50	23%	46%	0.87
	Flickr4Java	-0.44	0.84	0.87	0%	12%	0.39
	FXGL	-0.42	0.81	0.73	19%	26%	2.14
	GameComposer	2.59	3.58	0.82	13%	38%	8.26
	getting-started-java	-0.55	0.58	0.75	19%	26%	0.39
	IRI	-0.93	0.95	1.00	0%	7%	0.17
	jChecs	-0.10	0.46	0.34	41%	68%	0.53
	JFreeChart	-0.53	0.55	0.60	31%	46%	0.37
	jsonter	0.11	1.34	0.76	13%	16%	3.8
	Libresonic	2.10	2.88	1.00	15%	15%	5.19
	MovSim	-0.63	0.67	0.75	9%	15%	0.26
	MyBatis	-0.44	0.52	0.62	37%	45%	0.36
	Ninja	-0.35	0.85	0.80	14%	28%	0.9
	OkHttp	-0.04	0.59	0.50	28%	44%	0.66
	OpenAudible	-0.75	0.75	0.81	4%	13%	0.22
	RoaringBitmap	-0.22	0.35	0.00	53%	63%	0.64
	Traccar	-0.13	0.56	0.50	35%	48%	0.55
	TrackMate	-0.49	0.52	0.61	24%	38%	0.28
	VeraPDF	-0.47	0.62	0.63	17%	33%	0.36
	All projects (21/21)	-0.38	0.62	0.68	27%	36%	1.19
Critical	Apache PDFBox	1.40	1.90	1.70	0%	0%	0.99
	Computoser	0.70	1.31	0.62	42%	50%	3.01
	docker-maven-plugin	-0.07	0.56	0.67	30%	30%	0.32
	Flickr4Java	3.65	4.71	0.93	0%	0%	8
	FXGL	0.30	1.10	0.94	10%	15%	0.78
	GameComposer	3.63	3.63	3.63	0%	0%	-
	getting-started-java	-0.36	0.36	0.43	27%	55%	0.23
	IRI	-0.72	0.92	1.00	10%	10%	0.27
	jChecs	-0.01	0.64	0.50	33%	50%	0.76
	JFreeChart	-0.51	0.51	0.50	25%	50%	0.34
	jsonter	-0.44	0.63	0.66	17%	33%	0.29
	Libresonic	2.64	3.36	0.87	33%	33%	4.9
	MovSim	-0.87	0.87	0.97	0%	9%	0.21
	MyBatis	-0.32	0.34	0.27	46%	77%	0.32
	Ninja	0.14	1.18	0.86	25%	33%	1.38
	OkHttp	-0.26	0.26	0.26	0%	0%	0.01
	OpenAudible	-0.43	0.60	0.70	17%	25%	0.32
	Traccar	-0.31	0.37	0.36	50%	67%	0.19
	TrackMate	-0.75	0.75	0.76	0%	25%	0.26
	VeraPDF	0.21	0.76	0.57	29%	44%	1.24
	All projects (20/21)	-0.01	0.74	0.53	26%	43%	1.31
Blocker	Apache PDFBox	0.87	1.29	0.90	17%	33%	1.2
	Computoser	1.26	1.74	1.74	0%	50%	1.78
	docker-maven-plugin	0.13	0.22	0.18	75%	0%	0.21
	Flickr4Java	-0.82	0.82	0.82	0%	0%	-
	FXGL	-0.23	0.52	0.38	15%	59%	0.33
	GameComposer	3.13	3.13	3.13	0%	0%	-
	getting-started-java	-0.20	0.66	0.53	16%	18%	1.58
	IRI	-1.00	1.00	1.00	0%	0%	0
	jChecs	-0.20	0.20	0.20	0%	0%	0
	JFreeChart	-0.59	0.59	0.59	0%	0%	-
	Libresonic	10.76	11.10	11.10	0%	50%	15.22
	MovSim	-0.63	0.63	0.63	0%	0%	0
	MyBatis	-0.28	0.39	0.16	67%	67%	0.54
	Ninja	0.40	1.30	0.80	7%	7%	2.1
	OkHttp	0.39	0.83	0.34	0%	67%	0.86
	OpenAudible	-0.63	0.63	0.63	0%	0%	-
	Traccar	1.07	1.82	1.00	14%	29%	2.98
	TrackMate	-0.81	0.81	0.81	0%	0%	-
	VeraPDF	-0.27	0.96	1.00	0%	17%	0.58
	All projects (19/21)	0.15	0.98	0.53	16%	31%	2.26

levels (i.e., minor, major, critical, and blocker). The p-value returned by the Kruskal-Wallis test was equal to: 0.45 for $Mean(RE)$, 0.808 for $MMRE$, 0.864 for $MdMRE$, 0.031 for $PRED25$, 0.243 for $PRED50$, and 0.872 for $SdMRE$. This means that, only for $PRED25$, there is a significant difference ($\alpha = 0.05$) in the distributions of the values among the severity levels. Since we found a significant difference, we could perform a post-hoc analysis (i.e., a comparison between each pair of severity levels) to determine which severity levels significantly differ from one another. As is customary for the Kruskal-Wallis test, we used the Dunn's test to perform the post-hoc analysis [43]. We found three pairwise comparisons (out of six) for which the distributions of the $PRED25$ values differed significantly ($\alpha = 0.05$), namely: blocker vs. minor (p-values = 0.027); blocker vs. major (p-value = 0.007); and blocker vs. critical (p-value = 0.02). In these three pairwise comparisons, the $PRED25$ values for the blocker severity levels were always worse (see Table 6).

Summing up, the remediation time suggested by SonarQube is inaccurate, and overestimated, whatever the severity level is. Moreover, for those TD items whose severity level is blocker, the remediation time suggested by SonarQube is even more inaccurate.

Finally, we would like to remark out that, unfortunately, for the info severity level, only two projects had some fixed TD items. Therefore, we do not make any conclusion about the accuracy of the remediation time of info TD items.

5.4. $RQ_{2.3}$—What is the accuracy of TD remediation time with respect to different types?

As shown in Table 4, the most fixed TD items are code smells (2,897). The number of fixed vulnerabilities and bugs is 505 and 234, respectively.

In Table 7, we report the values of the accuracy metrics when considering the different types. For both code smells and vulnerabilities, the TD remediation time suggested by SonarQube tends to be overestimated since the overall $Mean(RE)$ values are negative in both cases (i.e.,-0.34 and -0.39, respectively). Moreover, as for the code smell type, 18 out of 21 projects are overestimated, while, as for the vulnerability type, 13 out of 20 projects are overestimated. Instead, when considering the bug type, the overall $Mean(RE)$ value is positive (0.40); however, we can observe individual $Mean(RE)$ values that suggest an overestimation for the majority of the projects (13 out of 20). That is, for some projects (e.g., GameComposer),

Table 7: TD remediation time accuracy for each project grouped by type.

Type	Project	Mean(RE)	MMRE	MdMRE	PRED25	PRED50	SdMRE
Bug	Apache PDFBox	0.42	1.06	0.88	14%	29%	0.95
	Computoser	0.97	1.59	0.49	18%	55%	3.15
	docker-maven-plugin	-0.03	0.37	0.50	33%	67%	0.32
	Flickr4Java	-0.44	0.79	0.84	0%	0%	0.25
	FXGL	-0.48	0.71	0.71	15%	15%	0.35
	GameComposer	3.99	4.07	0.46	13%	63%	8.17
	getting-started-java	-0.01	0.90	0.71	33%	41%	2.15
	IRI	-0.38	0.79	1.00	0%	17%	0.33
	jChecs	-0.02	0.35	0.23	50%	71%	0.32
	JFreeChart	-0.48	0.50	0.48	46%	55%	0.42
	jsonter	-0.26	0.75	0.66	22%	22%	0.59
	Libresonic	3.40	4.00	1.00	12%	12%	6.5
	MovSim	-0.88	0.88	0.92	0%	0%	0.12
	MyBatis	-0.24	0.48	0.60	46%	46%	0.44
	Ninja	0.41	1.23	1.00	19%	30%	1.89
	OkHttp	0.34	0.69	0.29	40%	70%	0.98
	OpenAudible	-0.39	0.39	0.33	40%	60%	0.38
	Traccar	2.04	2.04	0.65	17%	33%	3.24
	TrackMate	-0.42	0.52	0.45	13%	50%	0.32
	VeraPDF	-0.06	0.66	0.63	23%	46%	0.6
	All projects (20/21)	0.40	1.19	0.68	24%	38%	2.78
Vulnerability	Apache PDFBox	0.00	1.00	0.78	11%	22%	0.91
	Computoser	-0.73	0.73	0.75	0%	17%	0.17
	docker-maven-plugin	0.20	0.20	0.20	0%	0%	-
	Flickr4Java	-0.18	0.18	0.00	80%	80%	0.33
	FXGL	0.15	1.26	0.88	16%	27%	2.41
	GameComposer	-0.61	0.61	0.71	17%	25%	0.27
	getting-started-java	-0.67	0.67	0.74	2%	9%	0.14
	IRI	-1.00	1.00	1.00	0%	0%	0
	jChecs	-0.52	0.52	0.69	25%	25%	0.34
	JFreeChart	-0.59	0.59	0.67	25%	25%	0.41
	jsonter	-0.91	0.91	1.00	0%	0%	0.16
	Libresonic	2.12	2.28	1.00	14%	43%	3.24
	MovSim	-1.00	1.00	1.00	0%	0%	-
	MyBatis	-0.32	0.32	0.00	60%	60%	0.44
	Ninja	-0.04	0.96	0.82	14%	19%	1.03
	OkHttp	0.95	1.37	0.96	0%	25%	1.32
	OpenAudible	-0.84	0.84	0.87	0%	0%	0.1
	Traccar	3.91	3.91	3.91	50%	50%	5.53
	TrackMate	-0.49	0.49	0.63	33%	33%	0.3
	VeraPDF	0.33	1.75	0.91	0%	0%	2.87
	All projects (20/21)	-0.39	0.84	0.74	8%	15%	1.15
Code Smell	Apache PDFBox	0.28	0.89	0.53	28%	46%	1.52
	Computoser	-0.54	0.65	0.75	17%	22%	0.33
	docker-maven-plugin	0.02	0.63	0.52	25%	39%	0.7
	Flickr4Java	-0.07	1.07	0.80	11%	20%	2.49
	FXGL	-0.36	0.78	0.75	14%	29%	1.51
	GameComposer	-0.91	0.91	1.00	0%	0%	0.18
	getting-started-java	-0.51	0.53	0.53	19%	30%	0.36
	IRI	-0.84	0.93	1.00	9%	11%	0.34
	jChecs	0.00	0.49	0.25	48%	65%	0.81
	JFreeChart	-0.44	0.44	0.49	23%	54%	0.27
	jsonter	-0.14	1.02	0.73	14%	23%	2.82
	Libresonic	-0.91	0.91	0.91	0%	0%	0.07
	MovSim	-0.64	0.67	0.75	8%	15%	0.26
	MyBatis	-0.37	0.47	0.39	39%	54%	0.37
	Ninja	-0.29	0.88	0.73	17%	30%	1.28
	OkHttp	-0.10	0.68	0.62	7%	40%	0.63
	OpenAudible	-0.68	0.73	0.75	2%	10%	0.19
	RoaringBitmap	-0.22	0.35	0.00	53%	63%	0.64
	Traccar	-0.12	0.57	0.47	39%	53%	0.66
	TrackMate	-0.33	0.61	0.61	24%	34%	0.53
	VeraPDF	-0.28	0.65	0.63	21%	37%	0.56
	All projects (21/21)	-0.34	0.62	0.63	26%	38%	0.98

the estimated remediation time was strongly underestimated, thus affecting the overall $Mean(RE)$. To better determine whether the estimated remediation time was overestimated for bugs, as well as for vulnerabilities and code smells, we ran a one-tailed Mann-Whitney U (H_1 was: *spent remediation time was significantly less than estimated remediation time*) for any type of TD items. Whatever the type was, the test returned a p-value approximately equal to zero. This allows us to conclude that the remediation time suggested by SonarQube is overestimated regardless of the type of TD items.

The remediation time suggested by SonarQube is inaccurate whatever the type is—e.g., considering $PRED25$ and $PRED50$, we can see that most values are less than or equal to 25% and 50%, respectively, for any type. As compared to vulnerabilities and bugs, the estimated remediation time of code smells seems to be slightly more accurate (but still inaccurate)—e.g., for any accuracy metric, the overall value achieved by the code smells is better than, or equal to, that achieved by bugs and vulnerabilities. To verify whether there was a significant difference in the accuracy of the estimated remediation time due to the type of TD items, we used the Kruskal-Wallis test. In no case, we found a significant difference ($\alpha = 0.05$) since the p-value returned by the test was equal to: 0.08 for $Mean(RE)$, 0.479 for $MMRE$, 0.126 for $MdMRE$, 0.269 for $PRED25$, 0.057 for $PRED50$, and 0.566 for $SdMRE$.

Summing up, the remediation time suggested by SonarQube is inaccurate, and overestimated, for any type of TD items. Moreover, no significant difference has emerged in the remediation time SonarQube suggests when comparing the different types of TD items.

5.5. $RQ_{2.4}$—*What is the accuracy of TD remediation time with respect to different effort levels?*

Table 4 shows that most fixed TD items are trivial (2,492). The participants also fixed a good number of TD items whose effort level was easy (947) or medium (173). As for the other levels, the participants fixed few TD items or none. Due to the lack of sufficient data points for the sizeable, high, and complex effort levels, we did not include these effort levels in the analyses that follow, as well as in our conclusions.

The results on the accuracy of TD remediation time per effort level are shown in Table 8. The results for $Mean(RE)$ clearly indicate that the remediation time is overestimated for the easy and medium effort levels. As for the trivial level, the trend is not so clear. We thus used the one-tailed Mann-Whitney U test to ascertain that, within each considered effort level,

the spent remediation time was significantly less than estimated remediation time (H_1). The results indicate that, whatever the effort level is, the remediation time suggested by SonarQube is overestimated—the p-values for the Mann-Whitney U test were always approximately equal to zero. The question that now arises is whether the remediation time suggested by SonarQube is similarly overestimated for any effort level. To that end, we ran the Kruskal-Wallis test—H_1 was: *there is a significant difference in the distributions of values of $Mean(RE)$ among the effort levels (i.e., trivial, easy, and medium)*—, which indicated that there was a significant difference (the p-value was approximately equal to zero). The post-hoc analysis (i.e., Dunn's test) revealed three significant pairwise comparisons (out of three), namely: trivial vs. easy (p-value ≈ 0), trivial vs. medium (p-value ≈ 0), and easy vs. medium (p-value ≈ 0). This is because the $Mean(RE)$ values for the easy level are more negative as compared to the trivial one, while the $Mean(RE)$ values for the medium level are more negative as compared to both trivial and easy levels. In other words, the results suggest the following trend: the higher the effort level estimated to fix TD items, the more overestimated their remediation time is.

As for the other metrics, the remediation time is not accurate whatever the effort level is—e.g., for any level, most $PRED25$ values are less than or equal to 25%. Moreover, no huge difference among the trivial, easy, and medium levels seems to emerge by observing the $MMRE$, $MdMRE$, or $PRED25$ values; conversely, the $PRED50$ values for the medium level seem to be worse. We applied either the one-way ANOVA test or the Kruskal-Wallis test to assess H_1 (i.e., *there is a significant difference in the distributions of values, for the considered accuracy metric, among the effort levels*). Only for $MdMRE$, we could apply the one-way ANOVA test. The p-values for the $MMRE$ (0.244), $MdMRE$ (0.288), and $PRED25$ (0.562) metrics did not allow us to accept H_1 ($\alpha = 0.05$). Conversely, we could accept H_1 for $PRED50$ (p-value = 0.017), namely there is a significant difference in the distributions of the $PRED50$ values among the considered effort levels. This difference was due to the medium level as suggested by the Dunn's test we used to perform the post-hoc analysis. This is because two pairwise comparisons (out of three) were significant, both involving the medium level, namely: trivial vs. medium (p-value = 0.006) and easy vs. medium (p-value = 0.037). In both cases, the $PRED50$ values for the medium level are significantly worse as Table 8 suggests. We could accept H_1 for $SdMRE$ as well (p-value ≈ 0). The post-hoc analysis (i.e., Dunn's test) revealed three

Table 8: TD remediation time accuracy for each project grouped by effort level.

Effort Level	Project	Mean(RE)	MMRE	MdMRE	PRED25	PRED50	SdMRE
Trivial	Apache PDFBox	0.46	1.01	0.62	23%	41%	1.52
	Computoser	0.07	1.09	0.71	21%	29%	2.14
	docker-maven-plugin	0.14	0.66	0.50	26%	44%	0.8
	Flickr4Java	0.06	1.09	0.75	15%	24%	2.73
	FXGL	-0.36	0.80	0.76	13%	26%	1.04
	GameComposer	1.75	2.50	0.66	13%	33%	6.03
	getting-started-java	-0.40	0.50	0.53	28%	43%	0.69
	IRI	-0.78	0.95	1.00	5%	9%	0.33
	jChecs	0.02	0.48	0.22	51%	65%	0.83
	JFreeChart	-0.39	0.39	0.42	39%	61%	0.33
	jsonter	0.04	1.11	0.66	16%	25%	3.12
	Libresonic	4.12	4.67	1.00	0%	0%	6.93
	MovSim	-0.63	0.67	0.75	8%	15%	0.26
	MyBatis	-0.26	0.40	0.33	44%	65%	0.39
	Ninja	0.03	1.00	0.75	20%	30%	1.51
	OkHttp	0.29	0.81	0.62	21%	47%	0.9
	OpenAudible	-0.68	0.68	0.73	6%	15%	0.22
	RoaringBitmap	-0.07	0.27	0.00	66%	79%	0.74
	Traccar	0.58	1.00	0.25	39%	54%	1.85
	TrackMate	-0.27	0.60	0.54	24%	44%	0.57
	VeraPDF	-0.18	0.66	0.57	24%	41%	0.83
	All projects (21/21)	-0.20	0.68	0.63	27%	41%	1.31
Easy	Apache PDFBox	-0.17	0.77	0.57	24%	41%	1.16
	Computoser	-0.43	0.64	0.65	7%	36%	0.3
	docker-maven-plugin	-0.16	0.45	0.50	30%	40%	0.25
	Flickr4Java	-0.37	0.67	0.75	23%	31%	0.53
	FXGL	0.13	1.57	0.86	12%	15%	4.82
	GameComposer	-0.34	0.46	0.32	20%	60%	0.31
	getting-started-java	-0.67	0.67	0.74	1%	6%	0.14
	IRI	-0.81	0.83	1.00	19%	19%	0.34
	jChecs	-0.13	0.45	0.40	29%	71%	0.29
	JFreeChart	-0.67	0.71	0.70	14%	29%	0.33
	jsonter	-0.70	0.71	0.76	10%	14%	0.24
	Libresonic	1.36	1.98	0.98	20%	30%	2.95
	MovSim	-0.71	0.71	0.85	14%	24%	0.27
	MyBatis	-0.31	0.42	0.24	51%	55%	0.35
	Ninja	-0.24	0.92	0.80	13%	26%	0.88
	OkHttp	0.13	0.81	0.42	13%	50%	0.92
	OpenAudible	-0.61	0.75	0.83	7%	13%	0.27
	RoaringBitmap	-0.47	0.47	0.76	33%	37%	0.35
	Traccar	-0.45	0.48	0.49	33%	50%	0.35
	TrackMate	-0.47	0.54	0.68	27%	27%	0.29
	VeraPDF	-0.46	0.71	0.76	11%	27%	0.37
	All projects (21/21)	-0.50	0.68	0.74	14%	21%	1.04
Medium	Apache PDFBox	-0.46	0.46	0.69	44%	44%	0.37
	Computoser	-0.85	0.85	0.85	0%	0%	0.15
	docker-maven-plugin	-0.44	0.57	0.64	33%	33%	0.34
	Flickr4Java	-0.90	0.90	0.92	0%	0%	0.07
	FXGL	0.19	0.54	0.34	36%	77%	1.05
	getting-started-java	-0.82	0.82	0.82	0%	0%	0
	IRI	-1.00	1.00	1.00	0%	0%	0
	jChecs	-0.43	0.49	0.56	50%	50%	0.47
	JFreeChart	-0.57	0.57	0.59	33%	33%	0.36
	jsonter	-0.63	0.66	0.81	25%	25%	0.4
	Libresonic	-0.27	0.27	0.27	50%	0%	0.09
	MovSim	-0.92	0.92	0.94	0%	0%	0.06
	MyBatis	-0.49	0.53	0.66	25%	38%	0.3
	Ninja	-0.74	0.74	0.86	0%	10%	0.25
	OkHttp	-0.33	0.33	0.33	0%	0%	-
	OpenAudible	-0.79	0.79	0.75	0%	0%	0.15
	RoaringBitmap	-0.62	0.62	0.80	21%	24%	0.34
	Traccar	-0.48	0.48	0.66	40%	40%	0.46
	TrackMate	-0.54	0.54	0.50	20%	20%	0.35
	VeraPDF	-0.72	0.91	0.87	0%	5%	0.29
	All projects (20/21)	-0.56	0.68	0.81	17%	27%	0.48
Sizeable	OkHttp	-0.50	0.50	0.50	0%	0%	-
	All projects (1/21)	-0.50	0.50	0.50	0%	0%	-
Complex	GameComposer	-0.91	0.91	1.00	0%	0%	0.18
	jChecs	-1.00	1.00	1.00	0%	0%	-
	MyBatis	-0.88	0.88	0.93	0%	0%	0.06
	Traccar	-0.78	0.78	0.78	0%	0%	-
	TrackMate	-0.91	0.91	0.91	0%	0%	-
	All projects (5/21)	-0.89	0.89	0.93	0%	0%	0.09

Table 9: TD remediation time accuracy for the 20 most fixed coding rules.

Squid	Mean(RE)	MMRE	MdMRE	PRED25	PRED50	SdMRE	#Items
S1989	-0.67	0.67	0.74	2%	7%	0.14	274
S125	-0.56	0.57	0.75	23%	28%	0.33	253
db	-0.65	0.67	0.76	12%	18%	0.26	205
S112	-0.34	0.63	0.76	35%	39%	2.02	203
S3776	0.32	0.69	0.43	34%	54%	0.77	160
S1104	-0.02	1.04	0.77	16%	26%	1.78	135
S1117	-0.27	0.28	0	53%	74%	0.33	115
S1192	-0.55	0.56	0.53	10%	44%	0.25	106
S2629	-0.26	0.43	0.33	48%	52%	0.66	82
S1141	-0.6	0.65	0.77	4%	20%	0.19	80
S1191	-0.68	0.69	0.77	0%	1%	0.13	77
S3415	0.01	0.31	0.23	51%	87%	0.48	71
S116	-0.37	0.46	0.38	39%	52%	0.4	68
S106	-0.55	0.64	0.69	12%	35%	0.32	66
S4274	-0.44	0.48	0.63	28%	39%	0.34	61
S1066	-0.27	0.44	0.35	45%	55%	0.5	60
S1118	-0.7	0.7	0.84	20%	20%	0.36	56
S1172	-0.19	0.44	0.32	49%	59%	0.66	53
S1604	0.15	0.37	0	68%	70%	0.62	47
S1845	-0.5	0.52	0.53	4%	9%	0.12	46

significant pairwise comparisons (out of three), namely: trivial vs. easy (p-value ≈ 0), trivial vs. medium (p-value ≈ 0), and easy vs. medium (p-value ≈ 0). This is because the $SdMRE$ values for the easy level are smaller as compared to the trivial one, while the $SdMRE$ values for the medium level are smaller as compared to both trivial and easy levels. In other words, we can observe a lower variability when the effort level increases.

Summing up, the remediation time suggested by SonarQube is inaccurate whatever the effort level is. Moreover, for those TD items whose effort level is medium, the remediation time suggested by SonarQube is even more inaccurate. Finally, the remediation time SonarQube suggests is overestimated for any effort level despite the higher the effort level, the more overestimated the remediation time is.

5.6. $RQ_{2.5}$—*What is the accuracy of TD remediation time with respect to different coding rules?*

In Figure 4, we depict the amount of non-fixed and fixed TD items that concern the most violated coding rules in the considered projects. From this figure, we can observe that only a small fraction of introduced TD items were fixed by the participants to reach the established target values of TD, with the only exception of the rule *S1989*—274 of out 276 introduced TD items were fixed. All the introduced and fixed TD items for that rule are from the getting-started-java project.

The values of accuracy metrics for the 20 most fixed coding rules are shown in Table 9 (see Appendix A for a description of these rules). In total,

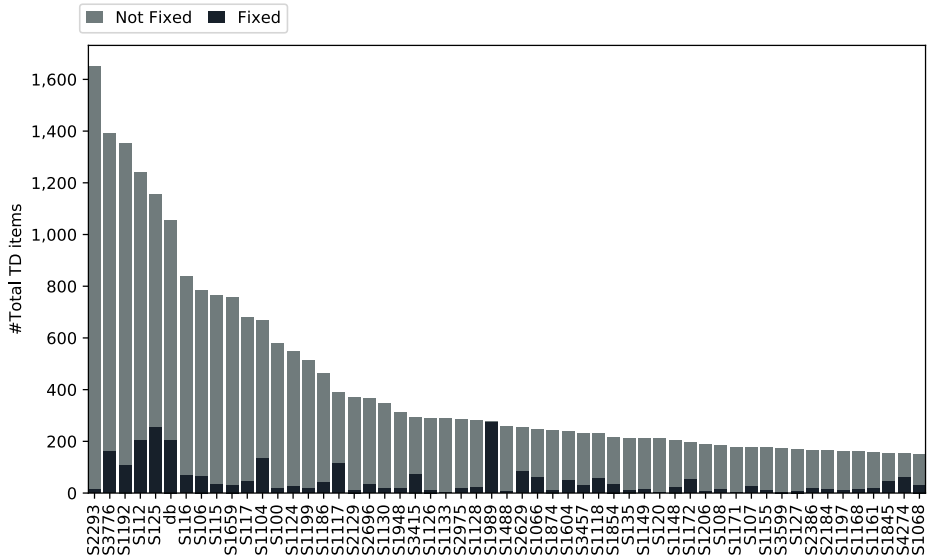

Figure 4: The number of non-fixed and fixed TD items for the most violated coding rules (in particular, the coding rules violated more than 150 times) in the analyzed projects.

the participants fixed TD items from 203 different coding rules of which four rules (i.e., S1989, S125, db, and S112) were fixed more than 200 times each, and other four rules (i.e., S3776, S1104, S1117, and S1192) were fixed between 100 and 200 times each. The coding rules fixed less than five times account for 49% (i.e., 100 out of 203).

For the 20 most fixed coding rules, the $Mean(RE)$ values suggest that the SonarQube's remediation time is generally overestimated (17 out of 20 coding rules). There is variability between the remediation time accuracy of the coding rules. For example, the rules *S1604*, *S3415*, and *S1117* are well estimated (e.g., see the $PRED25$ and $PRED50$ values). On the other hand, the rules *S1845*, *S1191*, and *S1989* are not well estimated (see $PRED25$ and $PRED50$ values under 10%).

6. Discussion

In line with Saarimäki et al.'s study [9], which investigated the diffuseness of TD items in 33 Java projects belonging to the Apache ecosystem, we found that: *(i)* TD items are diffused; *(ii)* most introduced TD items are code smells; and *(iii)* only a small proportion of introduced TD items is info or blocker. Our previous work [10], which we have extended here with six more

projects, supports the above-mentioned findings as well. That is to say that our results strengthen the external validity of both the work by Saarimaki et al. [9], since we considered a different sample of projects (i.e., we did not consider any projects among those considered by Saarimäki et al.), and our previous work [10], since we considered a larger amount of projects. The diffuseness of TD items in software projects, along with the claimed negative effects that TD items can have on those projects, should foster the research on models that accurately estimate the time to remedy TD items.

Out of the top ten violated coding rules we identified, five (i.e., db, S106, S112, S1192, and S3776) have been also listed, by Digkas et al. [8], among the top ten violated coding rules of the Apache ecosystem. We believe that these five coding rules deserve attention by researches, who should investigate the actual impact (not the claimed one) that leaving these rules violated can have during software maintenance and evolution.

We found that the remediation time estimated by SonarQube, in the considered projects, is inaccurate ($\mathbf{RQ}_{2.1}$)—in line with our previous paper [10]. Such an outcome also holds for any severity level ($\mathbf{RQ}_{2.2}$), type ($\mathbf{RQ}_{2.3}$), and effort level ($\mathbf{RQ}_{2.4}$) of fixed TD items, while the estimated remediation time for certain coding rules (e.g., S1604) is accurate ($\mathbf{RQ}_{2.5}$) as compared to others (e.g., S1989). This should motivate researchers to devise and then assess estimation models able to better estimate the time to remediate TD items (also because there is empirical evidence that TD items are diffused in open-source software projects). We also found that the remediation time of some categories of TD items (e.g., TD items whose severity level is classified as blocker or TD items whose effort level is classified as medium) is more inaccurate as compared to others. Therefore, researchers interested in devising better estimation models could prioritize the more inaccurate categories of TD items.

TD items fixed in the selected projects were removed with less effort than planned. That is, the estimated remediation time was usually higher than the actual time the participants spent to remove the TD items. Furthermore, it seems that the more the time estimated to remediate TD items, the greater the overestimation was. These findings should allow SonarQube users to make more informed decisions during project execution and resource management by keeping in mind that developers actually need less time than what SonarQube suggests to remediate TD items—especially when the time suggested by SonarQube is not trivial (i.e., more than ten minutes). We observed that, in the considered projects, the estimated time is, on average,

30% greater than the actual time. Nevertheless, we cannot suggest, based on the data collected, an adjustment value that SonarQube users, such as team leaders, can use to have an accurate estimated remediation time (e.g, by subtracting an offset to the SonarQube's estimated remediation time). It is worth noting that the participants of our case study were last-year undergraduate students, representative of junior developers, who refactored code of open-source projects they had not developed. This surely enforces and confirms the above-mentioned findings, as we believe that experienced developers, and especially those familiar with the source code affected by TD items to be fixed, would spend even less time to remove the same TD items. Such a speculation should foster researchers to replicate our study with both experienced developers and developers familiar with the source code affected by the TD items to be fixed.

Except for our previous work [10] (that we have extended here), the present work is, to the best of our knowledge, the first at assessing the accuracy of remediation time suggested by SonarQube. Therefore, although we have the merit of having gathered initial empirical evidence on the accuracy of SonarQube's suggested remediation time, at the same time, we foster researchers to gather further empirical evidence through replications.

Curiously, to reach the target values assigned to each project—i.e., A for any quality characteristic and a TD of at most 2-3 days (see Section 4.3)—, the participates mostly fixed major (20%) and blocker (17%) TD items. While the participants were forced to remove blocker TD items, as their resolution was a necessary condition to reach the target values, this constraint did not apply for major TD items that were, therefore, freely chosen among the TD items. It would be interesting to understand how experienced developers prioritize the TD items to be fixed among those identified by SonarQube.

7. Threats to Validity

In this section, we discuss potential threats to the validity of our results according to Yin [44]. We also illustrate the actions we adopted to mitigate these threats.

Construct Validity. We used the built-in quality gate of SonarQube, (i.e., sonar way) to identify TD items in the considered projects since practitioners are reluctant to customize the built-in quality gate and mostly rely on

the standard set of rules [16]. This might affect the diffuseness of introduced TD items.

We have tried to replicate the conditions of practitioners that use SonarQube although we are aware that the identification of some coding rules might not be accurate.

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.

The participants of our study formed mutually-exclusive teams, each of which worked on a single project—i.e., when studying the accuracy of remediation time, the projects are confounded with the teams. 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 the accuracy of SonarQube's remediation time, we believe that replications where 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 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 decrease when a participant fixes several TD items that belong to the same coding rule. Although such an effect is most likely also present in an actual scenario (the more an actual developer fixes TD items belonging to a given coding rule, the less the spent remediation time should be), it might have some effect on the obtained results. We are however confident that this reflects real cases.

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

The TD items considered in our analyses might threaten the generalizability of the results. In particular, when studying $\mathbf{RQ}_{2.2}$, we did not have enough data points for the info severity level (see Table 4). Similarly, when

studying $\mathbf{RQ}_{2.4}$, we did not have enough data points for the sizeable, high, and complex effort levels. 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 (although we observed an increasing trend as shown in Section 5.5). While we gather initial empirical evidence on the accuracy of SonarQube's remediation time, we highlight the need of further studies focusing, in particular, on a sample of TD items different from ours.

The participants in our study were last-year undergraduate students that can be considered junior developers as explained in Section 4.2. We are aware that the results might be influenced by the experience of the participants. So probably senior developers would spend less time removing the considered TD items and, consequently, increase the overestimation of the remediation time we observed. Moreover, the use of students has the advantage that they have a more homogeneous background (e.g., development experience) and are particularly suitable to gather initial evidence [45]. Given these motivations, the use of students should be considered appropriate, as also suggested in the literature [39, 45]. Further studies involving senior developers are, however, advised.

Reliability. We used standard Python packages and R packages to perform the statistical analyses since they ease the replication of the results and increase confidence on their quality. We fixed α at 0.05 regardless of the statistical hypothesis test we ran, including the post-hoc analyses. In other words, we decided not to adjust for multiple comparisons. This is because, from one hand, p-value adjustments reduce the chance of making type-I errors but, on the other hand, they increase the chance of making type-II errors [46]. Given the exploratory nature of our study—it is the first study at assessing SonarQube's remediation time—, adjusting for multiple comparisons is not desirable and leaving the p-values unadjusted is recommended [47]. However, we advice to bring further evidence thought replications.

8. Conclusion and Future Development

In this paper, we aim to understand the accuracy of Technical Debt (TD) remediation time estimations that SonarQube associates for fixing TD items, as well as the diffuseness of TD items. For this purpose, we designed and conducted a case study where we asked 81 junior developers, represented

by final-year undergraduate students (in Computer Science), organized in teams, to improve the quality of 21 open-source Java projects. This was done by choosing and fixing TD items identified by SonarQube, and tracking the time needed to fix the TD items. Afterwards, we compared the actual remediation time to SonarQube's estimated remediation time.

The results show that TD items are diffused in the 21 analyzed projects and most items are code smells. They also show that SonarQube's remediation time estimates are inaccurate and overestimated in comparison to the actual remediation time needed to fix the TD items.

We are confident that the results obtained from this study will motivate researchers to devise and assess remediation estimation models currently used to estimate the time needed to solve TD items, especially in the case where TD items are diffused in open-source software projects. Furthermore, our results may allow practitioners who use SonarQube to make more informed decisions regarding project execution and management with respect to remediation time estimations.

As future work, we foresee the replication of our study in industrial settings based on the suggestions of existing work [48, 49] considering a wider range of projects where SonarQube is integrated into the development process [50]. Replications should also introduce changes to our experimental setting (e.g., by studying the accuracy of TD remediation time when more participants are assigned the same TD items in the same projects). Furthermore, we plan to study the accuracy of TD remediation time that other tools, such as NDepend[10] or Sonargraph,[11] suggest.

Finally, we are investigating the supposed negative effects of TD issues with respect to the quality characteristics of SonarQube (i.e., maintainability, vulnerability, and security).

References

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

[2] T. Besker, A. Martini, R. E. Lokuge, K. Blincoe, J. Bosch, Embracing

[10] https://www.ndepend.com

[11] https://www.hello2morrow.com/products/sonargraph

technical debt, from a startup company perspective, in: International Conference on Software Maintenance and Evolution (ICSME), pp. 415–425.

[3] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162), Dagstuhl Reports 6 (2016).

[4] V. Lenarduzzi, A. Sillitti, D. Taibi., Analyzing forty years of software maintenance models, in: International Conference on Software Engineering Companion, ICSE-C '17, pp. 146–148.

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

[6] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering (2019) 1–39.

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

[8] G. Digkas, M. Lungu, A. Chatzigeorgiou, P. Avgeriou, The evolution of technical debt in the apache ecosystem, in: Software Architecture, Springer International Publishing, 2017, pp. 51–66.

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

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

[11] V. Lenarduzzi, N. Saarimäki, D. Taibi, Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study, arXiv:1908.11590 (2019).

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

[13] W. Cunningham, The wycash portfolio management system, in: OOPSLA '92.

[14] W. Li, 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) 1120 – 1128.

[15] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, J. Syst. Softw. 101 (2015) 193–220.

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

[17] A. Chatzigeorgiou, A. Manakos, Investigating the Evolution of Bad Smells in Object-Oriented Code, in: 2010 Seventh Int'l Conf. the Quality of Information and Communications Technology, IEEE, 2010, pp. 106–115.

[18] R. Peters, A. Zaidman, Evaluating the Lifespan of Code Smells using Software Repository Mining, in: 2012 16th European Conf. Softw. Maintenance and ReEng., IEEE, 2012, pp. 411–416.

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

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

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

[22] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[23] W. H. Brown, R. C. Malveau, H. W. S. McCormick, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1998.

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

[25] C. Seaman, Y. Guo, Measuring and monitoring technical debt, volume 82 of *Advances in Computers*, Elsevier, 2011, pp. 25 – 46.

[26] A. Aldaeej, C. Seaman, From lasagna to spaghetti, a decision model to manage defect debt, in: Proceedings of the 2018 International Conference on Technical Debt, TechDebt '18, ACM, New York, NY, USA, 2018, pp. 67–71.

[27] N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, Comparing four approaches for technical debt identification, Software Quality Journal 22 (2014) 403–426.

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

[29] F. A. F. I. Tollin, M. Zanoni, R. Roveda, Change prediction through coding rules violations, EASE'17, ACM, New York, NY, USA, 2017, pp. 61–64.

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

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

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

[33] T. J. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (1976) 308–320.

[34] M. Baldassarre, M. Piattini, F. Pino, G. Visaggio, Comparing iso/iec 12207 and cmmi-dev: Towards a mapping of iso/iec 15504-7, in: Proceedings - International Conference on Software Engineering, pp. 59–64.

[35] C. Pardo, F. Pino, F. García, M. Piattini, M. Baldassarre, A process for driving the harmonization of models, in: ACM International Conference Proceeding Series, pp. 51–54.

[36] M. Baldassarre, D. Caivano, G. Visaggio, Software renewal projects estimation using dynamic calibration, in: IEEE International Conference on Software Maintenance, ICSM, pp. 105–115.

[37] D. Caivano, Continuous software process improvement through statistical process control, pp. 288–293.

[38] P. Runeson, Using students as experiment subjects - an analysis on graduate and freshmen student data, 7th International Conference on Empirical Assessment in Software Engineering (2003).

[39] M. Höst, B. Regnell, C. Wohlin, Using students as subjects—a comparative study of students and professionals in lead-time impact assessment, Empirical Software Engineering 5 (2000) 201–214.

[40] S. Conte, H. Dunsmore, V. Shen, Software effort estimation and productivity, volume 24 of *Advances in Computers*, 1985, pp. 1 – 60.

[41] M. Jorgensen, Experience with the accuracy of software maintenance task effort prediction models, IEEE Transactions on Software Engineering 21 (1995) 674–681.

[42] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, A. Wessln, Experimentation in Software Engineering, Springer, 2012.

[43] O. J. Dunn, Multiple comparisons using rank sums, Technometrics 6 (1964) 241–252.

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

[45] J. Carver, L. Jaccheri, S. Morasca, F. Shull, Issues in using students in empirical studies in software engineering education, in: Proceedings of International Symposium on Software Metrics, METRICS '03, IEEE, USA, 2003, p. 239.

[46] R. J. Feise, Do multiple outcome measures require p-value adjustment?, Medical Research Methodology 2 (2002).

[47] A. D. Althouse, Adjust for multiple comparisons? it's not that simple, The Annals of Thoracic Surgery 101 (2016) 1644 – 1645.

[48] M. Baldassarre, D. Caivano, G. Visaggio, Empirical studies for innovation dissemination: Ten years of experience, in: ACM International Conference Proceeding Series, pp. 144–152.

[49] P. Ardimento, D. Caivano, M. Cimitile, G. Visaggio, Empirical investigation of the efficacy and efficiency of tools for transferring software engineering knowledge, Journal of Information and Knowledge Management 7 (2008) 197–207.

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

Table A.10: A description of the coding rules reported in Table 3 and Table 9.

Squid	Description	Type	Severity
db	Source files should not have any duplicated blocks.	Code smell	Major
S106	Standard outputs should not be used directly to log anything.	Code smell	Major
S112	Generic exceptions should never be thrown.	Code smell	Major
S115	Constant names should comply with a naming convention.	Code smell	Critical
S116	Field names should comply with a naming convention	Code smell	Minor
S125	Sections of code should not be commented out.	Code smell	Major
S1066	Collapsible "if" statements should be merged.	Code smell	Major
S1104	Class variable fields should not have public accessibility.	Vulnerability	Minor
S1117	Local variables should not shadow class fields.	Code smell	Major
S1118	Utility classes should not have public constructors.	Code smell	Major
S1141	Try-catch blocks should not be nested.	Code smell	Major
S1172	Unused method parameters should be removed.	Code smell	Major
S1191	Classes from "sun.*" packages should not be used.	Code smell	Major
S1192	String literals should not be duplicated.	Code smell	Critical
S1604	Anonymous inner classes containing only one method should become lambdas.	Code smell	Major
S1659	Multiple variables should not be declared on the same line.	Code smell	Minor
S1845	Methods and field names should not be the same or differ only by capitalization.	Code smell	Blocker
S1989	Exceptions should not be thrown from servlet methods.	Vulnerability	Minor
S2293	The diamond operator ("<>") should be used.	Code smell	Minor
S2629	"Preconditions" and logging arguments should not require evaluation.	Code smell	Major
S3415	Assertion arguments should be passed in the correct order.	Code smell	Major
S3776	Cognitive Complexity of methods should not be too high.	Code smell	Critical
S4274	Asserts should not be used to check the parameters of a public method.	Code smell	Major

Appendix A. Summary of Some Coding Rules

In this appendix, we summarize some SonarQube's coding rules (see Table A.10). In particular, we focus on the coding rules reported in Table 3 and Table 9, while the full list of coding rules can be found in our replication package.[2]