Do Static Analysis Tools Affect Software Quality when Using Test-driven Development?

Simone Romano
University of Salerno
Italy
siromano@unisa.it

Fiorella Zampetti
University of Sannio
Italy
fiorella.zampetti@unisannio.it

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

Massimiliano Di Penta
University of Sannio
Italy
dipenta@unisannio.it

Giuseppe Scanniello
University of Salerno
Italy
gscanniello@unisa.it

ABSTRACT

Background. Test-Driven Development (TDD) is an agile software development practice, which encourages developers to write "quick-and-dirty" production code to make tests pass, and then apply refactoring to "clean" written code. However, previous studies have found that refactoring is not applied as often as the TDD process requires, potentially affecting software quality.
Aims. We investigated the benefits of leveraging a Static Analysis Tool (SAT)—plugged-in the Integrated Development Environment (IDE)—on software quality, when applying TDD.
Method. We conducted two controlled experiments, in which the participants—92, in total—performed an implementation task by applying TDD with or without a SAT highlighting the presence of code smells in their source code. We then analyzed the effect of the used SAT on software quality.
Results. We found that, overall, the use of a SAT helped the participants to significantly improve software quality, yet the participants perceived TDD more difficult to be performed.
Conclusions. The obtained results may impact: *(i)* practitioners, helping them improve their TDD practice through the adoption of proper settings and tools; *(ii)* educators, in better introducing TDD within their courses; and *(iii)* researchers, interested in developing better tool support for developers, or further studying TDD.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Test-driven Development, Static Analysis Tool, Refactoring

ACM Reference Format:
Simone Romano, Fiorella Zampetti, Maria Teresa Baldassarre, Massimiliano Di Penta, and Giuseppe Scanniello. 2022. Do Static Analysis Tools Affect Software Quality when Using Test-driven Development?. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '22), September 19–23, 2022, Helsinki, Finland.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3544902.3546233

ESEM '22, September 19–23, 2022, Helsinki, Finland

ACM ISBN 978-1-4503-9427-7/22/09…$15.00
https://doi.org/10.1145/3544902.3546233

1 INTRODUCTION

Test-Driven Development (*TDD*) is an agile software development practice in which unit tests are written before production code [7]. More in detail, TDD promotes short cycles, composed of three phases each, to incrementally implement software functionality:

Red Phase. Write a unit test for a small chunk of functionality not yet implemented and watch that unit test fail;
Green Phase. Make that unit test pass as quickly as possible, committing whatever "sin" is necessary to do so, and watch all unit tests pass;
Refactor Phase. Refactor the code, thus remedy any sin previously committed just to make the unit test pass, and watch all unit tests pass.

The Red-Green-Refactor cycle is thus repeated until the functionality is completely implemented. When this happens, the TDD practitioner can tackle new functionality.

Advocates of TDD claim that this agile practice allows improving software quality as well as developers' productivity [19]. These claimed benefits have encouraged some software companies to adopt TDD, while others have been considering its adoption [58]. There is also a wide community around TDD—for example, the "TDD" tag on *Stack Overflow* has 5.2k watchers (in 05/2022). Nevertheless, the results about the claimed benefits, gathered in secondary studies, are not conclusive yet (*e.g.,* [24, 31]).

A key role in TDD is played by refactoring since it allows to constantly improve software quality, while preserving software external behavior [19]. Nevertheless, empirical evidence shows that the Refactor phase is often skipped (*e.g.,* [2, 48, 53]). For example, Romano *et al.* [48] observed that refactoring is not performed as often as TDD requires and the Refactor phase is (wrongly) perceived as less important than the other two (*i.e.,* Red and Green). Skipping the refactor phase is fine only when there is no refactoring opportunity. In any other case, skipping this phase would negatively affect software quality. This might explain inconclusive results on the claimed benefits of TDD on software quality [24, 31, 54].

What triggers actions aimed to improve software quality, including refactoring actions, both inside and outside the TDD process? On the one hand, developers may be driven by their own experience, thus they autonomously decide when the code becomes too complex, or when the initial design solution no longer fits the ongoing evolutionary changes. On the other hand, actions to improve software quality can be triggered by indicators—*e.g.,* code metrics or code smells—provided by Static Analysis Tools (SATs). There are various scenarios in which developers can leverage SATs [63]: *(i)* while coding, by highlighting warnings (like code smells) directly in IDEs (Integrated Development Environments); *(ii)* within a Continuous Integration (CI) pipeline [70], which could either produce a set of warnings or make a build fail if the code is not compliant with given coding guidelines; or *(iii)* during a code review by running a "bot" [3, 55].

This paper focuses on the first scenario mentioned above in the context of TDD. Specifically, we study the effect of using a SAT on software quality while developing software with TDD. To that end, we conducted two experiments where the participants had to perform an implementation task in Java by applying TDD. The participants were divided into two groups: (1) an intervention group leveraging *SonarLint* [56], a SAT intended as a plugin for the Eclipse IDE, configured to highlight code smells while coding; and (2) a control group not using SonarLint.

The first experiment (also referred to as **Exp1**) involved 68 3rd-year Bachelor's students in Computer Science (CS), having an established time (three hours and a half) to perform the task. This implies that: the participants could not "take their time" to think about proper design solutions or devote all the time they wanted to refactorings, and not all managed to complete the task. The second experiment (also referred to as **Exp2**) was a "differentiated" replication with 24 1st-year Master's students in CS. In this case, we did not give the participants a fixed time limit, hence allowing everybody to complete the task.

We measured whether the use (or not) of SonarLint produced significant differences in terms of several indicators, directly/indirectly related to software quality, namely: *(i)* number of code smells; *(ii)* estimate of technical debt; *(ii)* object-oriented code metrics; and *(iv)* code understandability and readability metrics. We found that, overall, the use of SonarLint helps developers to significantly improve code quality in terms of: fewer code smells, reduced technical debt, and better code understandability. However, the participants found TDD more difficult when using SonarLint.

Paper Structure. In Section 2, we describe the planning of the two experiments. In Section 3, we report the obtained results and then delineate the threats to their validity. We follow up with a discussion, outlining practical implications in Section 4. In Section 5, we overview existing literature on TDD and the use of SATs during software development, relating these studies to ours. We conclude the paper and outline directions for future work in Section 6.

2 PLANNING AND DESIGN

We report the planning of our experiments by using a template inspired by Jedlitschka *et al.*'s one [28]. When planning and executing our experiments, we followed the guidelines by Juristo and Moreno [30] and Wohlin *et al.* [68].

2.1 Goals

The overarching goal of our experiments, formulated on the basis of the *Goal Question Metric* template [5], is:

> **Analyze** the use of a SAT **for the purpose of** evaluating its effect in the TDD process **with respect to** software quality **from the point of view** of practitioners, educators, and researchers **in the context of** Bachelor's and Master's students in CS who develop software in Java.

According to our goal, we formulated and addressed the following Research Question (RQ):

RQ. *When applying TDD, is there a difference in software quality when using or not using a SAT?*

TDD encourages developers to write "quick-and-dirty" production code to make tests pass, and then apply refactoring to "clean" the written code [39]. Empirical evidence has shown that the Refactor phase is often skipped (*e.g.,* [2, 48, 53]), so potentially affecting, in a negative way, software quality. Therefore, we hypothesize that a SAT, like SonarLint, can lead developers to improve the quality of their code when necessary. Our RQ aims to verify such hypothesis.

2.2 Experimental Units

The participation in the experiments was voluntary, free, and, in line with Carver *et al.*'s suggestions [15], worth two bonus points on the final marks of the courses in which the experiments were embedded. Also, we informed the students that: *(i)* they would receive the bonus regardless of their performance in the experiment; *(ii)* they could drop out of the experiment at any time without being negatively judged; *(iii)* they could reach the highest mark of the course even without participating in the experiment; and *(iv)* the gathered data would be confidentially treated and anonymously shared for research purposes only.

Exp1. The participants were 3rd-year Bachelor's students in CS at the University of Bari. They were sampled by convenience among the students taking the *Integration and Testing* course. Among the students of this course (81), 68 accepted to participate in Exp1. The participants had a development experience ranging from 1 to 9 years (4.6 on average) and a Java development experience ranging from 0[1] to 8 years (2.4 on average). Only one participant had less than 1 year of experience in Java and indicated a knowledge level of 2 on a five-point Likert scale [42]. However, that participant had enough development experience as he/she was a professional developer. Finally, six participants were already professional developers (*i.e.,* part-time students).

Exp2. The participants were 1st-year Master's students in CS at the University of Sannio. They were sampled by convenience among the students taking the compulsory *Advanced Software Engineering* course (24 out of 30 students participated in the experiment). The participants in Exp2 were 24 with a development experience ranging from 2 to 13 years (4.7 on average) and a Java development experience ranging from 2 to 6 years (3.3 on average). Finally, two participants were professional developers.

Regardless of the experiment, the participants had refactoring and *JUnit* experience (raw data available in our online material [47]).

[1] We used an increment of 0.5 years to gather the years of experience. It is easy to grasp that if a participant had two months of experience, he/she rounded his/her years of experience to 0.

2.3 Experimental Material

The participants (in both Exp1 and Exp2) had to implement an API (Application Programming Interface), named *MRA* (*Mars Rover API*), for moving a rover on a planet. The planet is represented as a grid, where the cells can contain obstacles that the rover cannot pass through. MRA allows initializing the planet and moving the rover on the planet. The rover moves by parsing a string of commands. Once the rover moves, it returns its current position and orientation, along with the obstacles it encountered (if any). MRA is popular in the agile community, where it is used as a code kata (*i.e.,* a programming exercise used to hone developers' skills). Moreover, MRA has been used as an experimental object in several empirical studies on TDD with students and professionals (*e.g.,* [21, 23, 32, 58]). To deal with threats to *experimenter expectancies* [68], we reused the experimental material by Fucci *et al.* [23].

The experimental material includes the description of the functionality to be implemented and the project template (for the *Eclipse* IDE) containing method stubs (*i.e.,* empty methods exposing the expected API signatures) and an example JUnit test class.

The participants (in both Exp1 and Exp2) implemented, for training purposes (*i.e.,* before implementing MRA), two other APIs—*Pig Latin Translator* (*PLT*) and *Bowling Score Keeper* (*BSK*)—following TDD. Both PLT and BSK are popular code katas, already used in past studies (*e.g.,* [21, 22, 32]).

2.4 Experimental Task

In both experiments, we had a single task consisting of implementing MRA's functionality, decomposed into 11 user stories, by applying TDD. That is, the description of MRA's functionality was *fine-grained*—as opposed to a *coarse-grained* description where the functionality is described as a whole without an explicit decomposition into user stories [32]. A fine-grained decomposition allowed us to better monitor participants' behavior.

2.5 Variables, Measurements, and Hypotheses

The participants (in both Exp1 and Exp2) were asked to carry out the experimental task by using or not a SAT. In particular, the participants administered with the SAT had to follow TDD exploiting the SAT during the Refactor phase. The participants not using the SAT had to follow TDD to develop MRA and refactor their code (in the Refactor phase) without any SAT support (*i.e.,* relying just on their ability to identify refactoring opportunities).

Among the available SATs, we choose SonarLint for three main reasons. First, we are interested in experimenting with a SAT that identifies code quality problems like code smells. Second, SonarLint highlights code smells (and also other issues we disabled) while coding. That is, it raises a warning in the IDE when the developer has just introduced a code smell. SonarLint is not invasive and the developer is free to choose whether to ignore or address a warning. Therefore, SonarLint is suitable for use in the TDD process: the developer can ignore any warning in the Red and Green phases, and then address the warnings in the Refactor one. Third, SonarLint uses the same code-smell detector as *SonarQube*—a SAT popular in both industrial and open-source contexts [38, 64]—yet the former works within IDEs.

In our experiments, we thus have one independent variable (also known as main or manipulated factor): **Treatment**. It is a nominal variable assuming two values: *SonarLint* and *NoSonarLint*.

To quantify the software quality construct (in both Exp1 and Exp2), we considered the dependent variables that follow.

#Smell. It is the number of code smells detected by *SonarQube* in the MRA codebase each participant developed. Code smells are indicators of refactoring opportunities to improve software quality [20]. #Smell assumes values in $[0, +\infty]$. The higher the value, the worse the software quality is. While it is obvious that developers using SonarLint have a direct guidance towards removing such smells, we measured this variable also to understand, as a sanity check, whether the use, or not, of the SAT changes at all the number of smells in the codebases.

SqaleIndex. It is a technical debt estimate measured as the estimated time to remove all code smells from a codebase based on the *SQALE* (*Software Quality Assessment Based on Lifecycle Expectations*) method [37]. Such method is used nowadays by a number of SATs including, among others, SonarQube, *SQuORE*, and *NDepend*. We used SonarQube to compute SqaleIndex, which assumes values in $[0, +\infty]$. The higher the value, the higher the technical debt is (thus the worse the software quality is).

WMC. It is the *Weighted Methods Per Class* metric averaged across the classes a participant wrote. Given a class C, WMC_c is the sum of the cyclomatic complexity of the methods in C. WMC_c is one of the *CK metrics*, proposed by Chidamber and Kemerer [16], to assess software quality in the object-oriented context. We used the tool by Aniche [1] to compute, for each class written by a participant, WMC_c. We then averaged across the written classes to compute WMC; this is because we needed a single data point per participant. The WMC values range in $[0, +\infty]$; the higher the value, the higher the complexity of the code is.

CognCompl. It is the *Cognitive Complexity* metric [14], computed by SonarQube, averaged across the classes a participant implemented. Cognitive Complexity aims to measure code understandability. Empirical evidence has shown that Cognitive Complexity is a reliable indicator of code understandability since it positively correlates with comprehension time and subjective understandability [40]. CognCompl assumes values in $[0, +\infty]$; the higher the value, the worse the code understandability is.

BW. It is the *Buse-Weimer* metric [13] averaged across the classes a participant wrote. This metric was validated to measure code readability [13]; however, it is not suitable to capture code understandability [51]. To gather the *Buse-Weimer* metric, we used the tool Buse and Weimer made available on the web. BW assumes values in $[0, 1]$; the higher the value, the better the code readability is.

Besides the above-mentioned dependent variables, we also considered **CBO**, **RFC**, and **LCOM**. These dependent variables are the CK metrics [16]—*i.e., Coupling Between Objects*, *Response For a Class*, and *Lack of COhesion in Method*, respectively—averaged across the classes written by a participant. Since the implementation of MRA did not require the use of hierarchies, we did not consider the remaining two CK metrics: *Depth of Inheritance Tree* and *Number of Children*. We also considered the dependent variable **LOC**, which is the *Lines of Code* metric averaged across the classes written by a participant. The higher the values of CBO, RFC, LCOM, and LOC,

the worse it is. For space reasons, we chose to confine the results about CBO, RFC, LCOM, and LOC to our online material [47].

Summing up, we used a number of dependent variables to measure software quality; each one capturing different aspects of software quality (*e.g.,* from code complexity to code readability). This is to mitigate a *mono-method-bias* threat [68] and to have a clearer and wider picture of the phenomenon under investigation.

To answer RQ, we formulated and tested, for each dependent variable X, the following null hypothesis:

$H0_X$. There is no statistically significant difference, with respect to X, between SonarLint and NoSonarLint.

2.6 Study Design

We used, for both Exp1 and Exp2, the *one factor with two treatments* experimental design [68], where the two treatments are: **SonarLint** and **NoSonarLint**. The former represents the intervention group, while the latter represents the control group. The used design is a kind of *between-subjects* design; each participant is assigned to only one treatment (*i.e.,* the participant either used SonarLint or not).

When planning the experiments, we discarded alternative designs, such as *repeated measures* (where each participant experiments with both treatments), because they are more vulnerable to threats to validity prominent in our context. In particular, *learning* and *carryover* effects can interfere with the results when participants' performance is measured under one experimental condition (*e.g.,* SonarLint) and later under the other (*e.g.,* NoSonarLint). Controlling or compensating for such effects is risky as it implies the use of statistical techniques that might hamper the interpretation of the results [22, 35, 65].

In each experiment, the participants were randomly assigned to either the intervention or control group. With randomization, the participants have the same chance to be assigned to the intervention and control groups. Randomization is an important design principle that helps produce comparable experimental groups by mitigating the effect of unwanted factors [68]. We assigned 34 participants to each experimental group (treatment and control) in Exp1, and 12 to each experimental group in Exp2. That is, the design of Exp1 and Exp2 was also balanced. This is desirable in SE experiments because it strengthens statistical analyses [68].

2.7 Procedure

The procedure we followed in Exp1 and Exp2 consisted of the following steps.

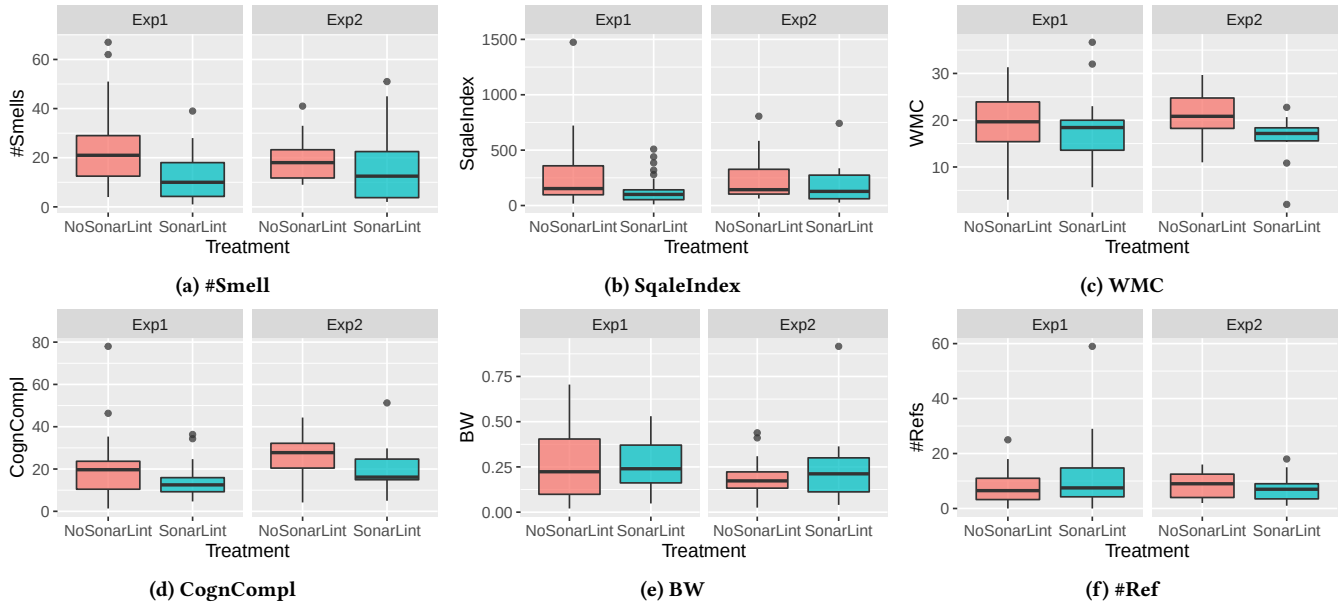
Recruitment. We collected students' adhesion to the experiment with a *Google Forms* pre-experiment questionnaire. With such questionnaire, we also gathered some demographic information about the participants. Specifically, we asked them whether they had (1) any professional development experience, and (2) the years of experience as well as the self-assessed expertise (over a 5-levels Likert scale [42]) about development in general, Java, testing, JUnit, and refactoring.

Training. All the participants attended a frontal lecture on TDD. Right after, the participants take part in a laboratory session in which they were asked to develop the functionality of *PLT* by following TDD. PLT's functionality included eight user stories. Five user stories were implemented with the assistance of the lecturer who interacted with the participants to show them how to use TDD to implement those user stories. The remaining three user stories were implemented by the participants alone. To implement PLT's functionality, the participants used Eclipse as the IDE to write source code, its JUnit plugin to run tests, and Git (either from the command line or through the Eclipse plugin). We instructed the participants to commit, at least, each time their test suite passed (*i.e.,* end of the Green and Refactor phases). At the end of the development task, the participants pushed their local changes to their remote Git repository.

Assignment. The participants were randomly split into the experimental groups (*i.e.,* SonarLint and NoSonarLint). Each group participated in a warm-up session. The participants in the SonarLint group were provided with the same Eclipse distribution (based on the 2020-06 version) in which SonarLint was already plugged-in and pre-configured. Specifically, we used a customized version of the built-in *Sonar-way* quality profile to configure SonarLint. Sonar-way is the default quality profile and is used to set which issues—*i.e.,* bugs, vulnerabilities, and code smells—SonarLint has to detect. Since bugs and vulnerabilities are not of our interest (given our research goal), we disabled them so that SonarLint only highlighted the presence of the code smells specified by Sonar-way. We based our SonarLint configuration on a default quality profile because developers are usually reluctant to customize SATs [64]. As for the participants in the NoSonarLint group, they were equipped with the same Eclipse distribution as the other group but without SonarLint. Regardless of the group, the participants had to implement (alone) BSK's functionality (consisting of 12 user stories) by applying TDD (*i.e.,* the only difference was represented by SonarLint). To do so, the participants had to use the provided Eclipse distribution—we forbid the participants to install any other plugin.

Operation. The participants in the SonarLint and NoSonarLint groups took part in the experimental session at the same time. The procedure and tools used were the same as the warm-up session. The participants were asked to implement (individually, without collaborating) MRA's functionality by applying TDD. At the beginning of the experimental session, each participant was provided with: *(i)* the description of MRA's functionality, structured in 11 user stories, and *(ii)* the project template for Eclipse, hosted on a Git remote repository. The participants had to fork the Git remote repository and then clone it. Once imported the project template into Eclipse, they could start tackling the user stories once at a time, by using the Eclipse distribution we provided them during the warm-up session. Whatever the group was, the participants were instructed to commit (at least) each time the test suite passed. At the end of the task, the participants sent their (local) commits to their remote repository and then filled in a post-experiment questionnaire in which they provided a link to their remote repository and answered some questions, on a 5-level Likert scale [42], about their perception of: Q1) user stories understanding, Q2) task ease, Q3) TDD ease, Q4) refactoring ease, and Q5) refactoring usefulness.

The aforementioned steps were the same in both experiments with just one exception: in Exp1, we fixed a time limit of three hours and a half to complete the experimental task; in Exp2, we did not give the participants a time limit. This design choice was to observe possible differences on if the time limit could play a role when the participants had to refactor their code. While the



(a) #Smell (b) SqaleIndex (c) WMC

(d) CognCompl (e) BW (f) #Ref

Figure 1: Boxplots for each dependent variable and treatment in each experiment, including #Ref.

experimental session of Exp1 took place in a laboratory setting, the one of Exp2 was conducted in a hybrid setting. That is, the participants in Exp2 started working in a laboratory setting and then continued working at home until completing the task.

2.8 Analysis Procedure

We analyzed the gathered data based on the following procedure.

Individual Analyses. For each experiment and treatment, we computed descriptive statistics (available in our online material [47]) and built boxplots to depict the distributions of the values of each dependent variable. Then, we tested the null hypotheses within each experiment. We planned to use consistent statistical methods across the experiments in order to exclude differences caused by the use of different statistical methods [50]. Specifically, given a dependent variable, if the normality assumption held in both Exp1 and Exp2, we used the (unpaired two-tailed *Welch's*) *t-test* to test the null hypotheses; otherwise, we used its non-parametric alternative, the (unpaired two-tailed) *Mann–Whitney U test*. To check the normality assumption, we used the *Shapiro-Wilk test*. We fixed the significance level (α) at 0.05.

Aggregate Analyses. We used two statistical methods to provide joint conclusions across the two experiments. The former statistical method is *Aggregated Data* (*AD*), also known as meta-analysis of effect sizes. There are two kinds of meta-analysis models: *fixed-* and *random-effects* [12]. Santos *et al.* [50] recommended the use of random-effects meta-analysis models in SE research. We followed this recommendation and fed the random-effects meta-analysis models (one for each dependent variable) with the *Standardized Mean Difference* (*SMD*) computed for each experiment. SMD is a standardized effect size measure. To estimate SMD, we used *adjusted Hedges' g*. The statistical significance of AD is decided by whether the 95% Confidence Interval (CI) of the joint effect size does not cross 0. As for the latter statistical method, we used *Stratified Individual Participant Data* (*IPD-S*). Given a dependent variable, IPD-S consists of jointly analyzing the data from all experiments by acknowledging which experiment the data come from. There are two kinds of IPD-S models: *fixed-* and *random-effects* [67]. Santos *et al.* [50] recommended the use of random-effects IPD-S models in SE research. Commonly used random-effects IPD-S models are *Linear Mixed Models* (*LMMs*) with two factors: *Experiment* and *Treatment* [67]. Accordingly, we used LMMs and modeled Experiment and Treatment as the two factors. The joint effect of Treatment is deemed statistically significant if the p-value is less than α (0.05).

As an additional analysis, to understand how the participants improved software quality, we detected the number of refactorings (**#Ref**) performed by the participants and the type thereof. To that aim, we ran the state-of-the-art *RefactoringMiner* tool [62] over each commit in the participants' repositories.

3 RESULTS

In this section, we present the results from the individual analyses followed by those from the aggregated ones. Then, we show the results from the additional and post-experiment questionnaire analyses. We conclude the section by outlining the threats to validity.

3.1 Individual Analysis Results

Fig. 1 shows the boxplots depicting the distributions of the values of the dependent variables, for each experiment and each treatment. As for #Smell, SqaleIndex, WMC, and CognCompl in Exp1, we can notice that the boxes for SonarLint are lower or slightly lower than the boxes for NoSonarLint. It seems that, in Exp1, the codebase of the participants using SonarLint is less smelly, has a lower technical debt estimate, is less complex, and is easier to understand as compared to the participants not equipped with SonarLint. These

trends are somewhat confirmed in Exp2 where, however, the difference between SonarLint and NoSonarLint is less pronounced (see Fig. 1.a, Fig. 1.b, and Fig. 1.d) with one exception, WMC. Indeed, the difference between SonarLint and NoSonarLint for WMC is larger in Exp2 (see Fig. 1.c). As for BW, the boxes of SonarLint and NoSonarLint mostly overlap one another in both experiments suggesting that there is not a huge difference in terms of code readability between the two experimental groups.

Table 1 reports the hypotheses testing results for the two experiments being considered individually. Since, for any dependent variable, the normality assumption did not hold in both experiments, we always used the Mann-Whitney U test (see Section 2.8). The results concerning the normality testing are available online [47]. Table 1 also reports, for each dependent variable and each experiment, *Cliff's* δ [25], a non-parametric effect size.[2]

As for Exp1, we found statistically significant differences for three out of five dependent variables, namely: #Smell (p-value = 0.001), SqaleIndex (p-value = 0.009) and CognCompl (p-value = 0.037). These significant differences were all in favor of SonarLint (see the δ values in Table 1 and/or the boxplots in Fig. 1). The effect size is medium for #Smell and SqaleIndex, while it was small for CognCompl. This implies that the availability of SonarLint, in Exp1, helped significantly reduce the spread of code smells. This outcome is quite expected since SonarLint was configured for detecting code smells and the configuration was the same as SonarQube—the tool used to measure #Smell. Also, SonarLint helped significantly reduce the technical debt estimate and improve code understandability.

As for Exp2, the only statistically significant difference concerned WMC (p-value = 0.022), such difference was in favor of SonarLint (*e.g.,* see the δ value in Table 1) and the effect size was large. Accordingly, the use of SonarLint in Exp2 helped significantly write less complex code.

Summing up, although we can observe differences in the statistical conclusions of Exp1 and Exp2 for a given dependent variable, the trend observed in an experiment is roughly confirmed by the trend observed in the other experiment for that dependent variable (*i.e.,* at most, the positive effect of SonarLint on a dependent variable is not so strong to be deemed significant).

3.2 Aggregate Analysis Results

Fig. 2 shows the AD results through forest plots (one for each dependent variable). The forest plots report, besides the SMD estimates and the corresponding 95% CIs, two statistics to assess between-study heterogeneity: the p-value from the *Q-test* and the I^2 statistic [12]. The Q-test allows determining whether there is a (statistically) significant between-study heterogeneity. The most used significance level is 0.1 [52]; therefore, if the Q-test p-value was lower than 0.1, we assumed that the between-study heterogeneity was significant. On the other hand, the I^2 statistic is used to measure the extent of between-study heterogeneity[3] when the Q-test p-value is less than the fixed significance level [52]. In Table 2, instead, we report the IPD-S results for each dependent variable—in particular, the LMM estimates (and their 95% CIs) for SonarLint with the corresponding p-values.

Table 1: Hypothesis testing results per experiment (in bold p-values < α), including Cliff's δ. The last row concerns #Ref.

Dep. Var.	Exp1		Exp2	
	p-value	(Cliff's) δ	p-value	(Cliff's) δ
#Smell	**0.001**	0.473 (medium)	0.248	0.285 (small)
SqaleIndex	**0.009**	0.369 (medium)	0.355	0.229 (small)
WMC	0.146	0.206 (small)	**0.022**	0.556 (large)
CognCompl	**0.037**	0.295 (small)	0.141	0.361 (medium)
BW	0.476	-0.102 (negligible)	0.755	-0.083 (negligible)
#Ref	0.246	-0.164 (small)	0.536	0.159 (small)

As shown in Fig. 2, when aggregating the data from both experiments, there are statistically significant differences between NoSonarLint and SonarLint with respect to #Smell, SqaleIndex, and CognCompl. This is because the CIs for "Overall" (indicating the joint effect size) do not cross 0 for these dependent variables. The joint effect size for #Smell is medium[4] and in favor of SonarLint. This is confirmed by the results shown in Table 2: the p-value (0.001) for #Smell is significant and the estimate is equal to -8.978—*i.e.,* the code base of the participants in the SonarLint group contains, on average, 8.978 code smells less than the control group. As for SqaleIndex and CognCompl, the joint effect size is, respectively, equal to -0.468 (see Fig. 2.b) and -0.497 (see Fig. 2.d), thus small and in favor of SonarLint in both cases. The IPD-S results in Table 2 confirm that there are significant differences, both in favor of SonarLint, with respect to SqaleIndex (p-value = 0.024) and CognCompl (p-value = 0.017). In other words, the participants equipped with SonarLint introduced significantly less technical debt, and their code was significantly easier to understand. As for WMC and BW, the AD and IPD-S results do not show statistically significant differences.

Finally, despite the change in the experimental setting (fixed time in Exp1 vs. non-fixed time in Exp2), we never find a statistically significant between-study heterogeneity as no Q-test p-value (see Fig. 2) is lower than 0.1. We can thus strengthen our overall statistical conclusions on the effect of SonarLint on software quality, especially, in terms of number of smells, technical debt estimate, and code understandability. As for the complexity of the written code (*i.e.,* WMC), we need further replications to reach the overall statistical conclusion that SonarLint helps write less complex code.

3.3 Additional Results

To understand how the participants improved code quality, we show the boxplots depicting the number of refactorings (#Ref) the participants performed in Exp1 and Exp2 in Fig. 1.f. In Table 1 (see the last row), instead, we show the results of the inferential analysis, computed using the Mann-Whitney U test since data were not normally distributed.

As shown in Fig. 1.f, there is not a large difference between SonarLint and NoSonarLint in both Exp1 and Exp2, despite two different trends seem to emerge. Specifically, in Exp1, the number of refactorings seems to be slightly greater when using SonarLint—this

[2]Based on Romano *et al.*'s guidelines [46], Cliff's δ is: *negligible*, if $|\delta| < 0.147$; *small*, if $0.147 \leq |\delta| < 0.33$; *medium*, if $0.33 \leq |\delta| < 0.474$; or *large*, otherwise.

[3]According to Higgins *et al.*'s guidelines [26], between-study heterogeneity is: *unimportant*, if $I^2 \leq 40\%$; *moderate*, if $30\% \leq I^2 \leq 60\%$; *substantial*, if $50\% \leq I^2 \leq 90\%$; or *considerable*, if $I^2 \geq 75\%$.

[4]According to Cohen's guidelines [17], SMD is: *negligible*, if |SMD| < 0.2; small, if $0.2 \leq$ |SMD| < 0.5; *medium*, if $0.5 \leq$ |SMD| < 0.8; or *large*, otherwise.

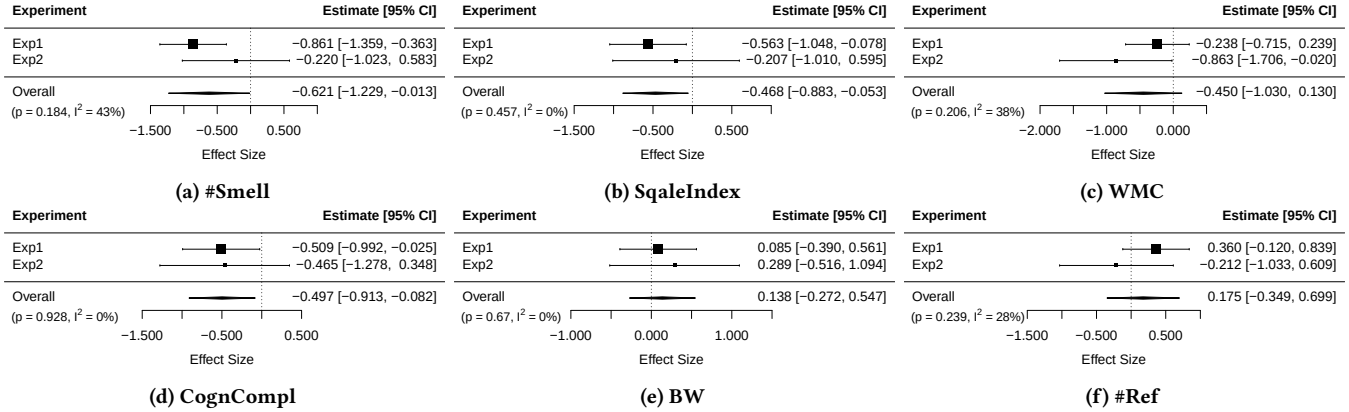


Figure 2: Forest plots showing the AD results for each dependent variable, including #Ref.

Table 2: IPD-S Results (in bold p-values < α). The last row concerns #Ref.

Dep. Var.	Estimate [95% CI]	p-value
#Smell	-8.978 [-14.319, -3.637]	**0.001**
SqaleIndex	-103.239 [-192.409, -14.069]	**0.024**
WMC	-2.463 [-5.134, 0.209]	0.07
CognCompl	-5.939 [-10.776, -1.102]	**0.017**
BW	0.025 [-0.043, 0.092]	0.472
#Ref	2.132 [-1.228, 5.493]	0.211

is quite in line with our expectations (*i.e.,* the use of SonarLint leads a developer to perform more refactorings). In Exp2, the number of refactorings seems to be roughly the same for both experimental groups. Nevertheless, we observed some improvements with respect to software quality in Exp2, especially for WMC. We thus inspected the types of refactorings the participants performed. In particular, we noticed that, in Exp2, the participants in the SonarLint group performed, overall, 17 *Extract Method* operations (*i.e.,* the refactoring type more likely to affect WMC), while the participants in the NoSonarLint group performed, overall, 25 such operations—as opposed to Exp1, where the overall *Extract Method* operations were 103 with SonarLint and 74 without SonarLint. We conjecture that the participants using SonarLint in Exp2 fixed the code smells in real time (*i.e.,* right after SonarLint highlighted them in the IDE) in the Green phase, making the refactoring operations not visible to RefactoringMiner.

The results from the inferential analysis in Table 1 do not indicate the presence of any statistically significant difference in Exp1 and Exp2 with respect to #Ref. This statistical conclusion is confirmed when aggregating the data from Exp1 and Exp2. Indeed, both AD (see Fig. 2.f) and IPD-S results (see the last row in Table 2) do not suggest the presence of a significant difference in the number of refactorings the participants performed when provided or not with SonarLint.

As mentioned in Section 2.5, we did not report detailed results for CBO, RFC, LCOM, and LOC due to space reasons—the interested reader can find these results in our online material [47]. For all the aforementioned dependent variables, and in both experiments (as well as by aggregating the data from both these experiments with AD and IPD-S), we did not find statistically significant differences between SonarLint and NoSonarLint. We only noticed, as expected, slightly lower LOC values for SonarLint in both Exp1 and Exp2.

3.4 Post-experiment Questionnaire Results

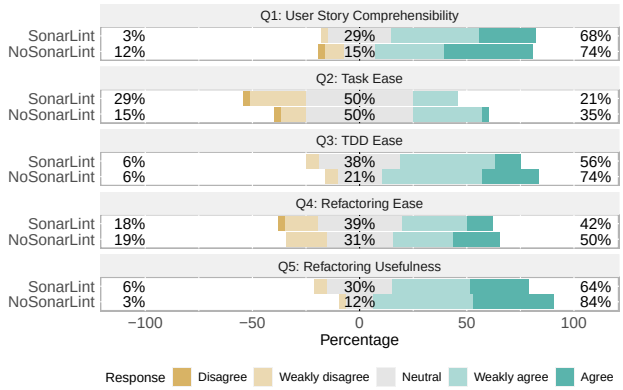
Fig. 3 shows diverging stacked bar charts summarizing the results of the post-experiment questionnaires, dividing the responses by SonarLint and NoSonarLint. The responses were also compared through the (unpaired two-tailed) Mann–Whitney U test; yet no statistically significant difference was found if analyzing the two experiments individually. On the overall dataset, IPD-S resulted in a significant difference for the ease of TDD (p-value = 0.03), and a marginally significant difference for the ease of the overall task (p-value = 0.059). In both cases, the difference is in favor NoSonarLint and confirmed by AD.

In both experiments, the majority of participants agreed about the comprehensibility of the user stories to be implemented, with percentages of agreement of 68% (for SonarLint) and 74% (for NoSonarLint) in Exp1, and 83% (for both cases) in Exp2. We can state that, at least based on what perceived by our participants, the understandability of the user stories did not introduce threats in the experiments.

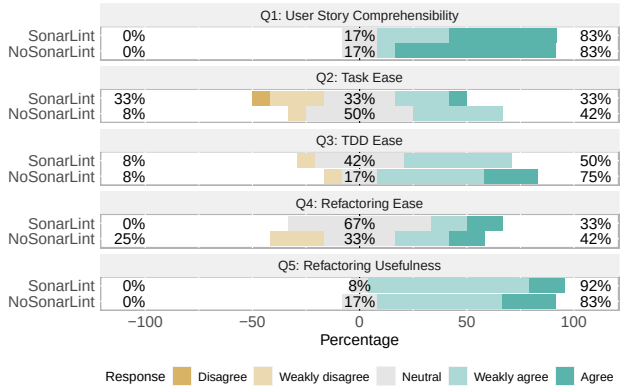
As for the task ease, we can notice that the percentages of agreement are relatively low, especially in Exp1, where half of the respondents perceived the task ease as neutral. We can also notice higher percentages of disagreement for the SonarLint group in both experiments.

TDD was considered ease by 56%-74% of the participants in Exp1 and 50%-75% in Exp2. The participants considered TDD easier when SonarLint was not used and, as said, the difference is statistically significant for the overall dataset. A possible interpretation is that the need for interpreting and addressing SonarLint warnings added an extra burden to the application of TDD, making it a little bit more difficult.

In Exp1, refactoring was considered easy by 42% and 50% of the participants, while the percentages are 33%-42% in Exp2. For Exp2, it is possible to notice a very high percentage of "Neutral" (67%) for the SonarLint treatment. In other words, all the participants in



(a) Exp1



(b) Exp2

Figure 3: Post-experiment questionnaire results.

the SonarLint group were either neutral or in agreement about the refactoring ease. In summary, especially in Exp2, the participants had a slightly less negative perception of refactoring ease when SonarLint was available. However, in general, refactoring is not necessarily perceived easier when a tool support (SonarLint in our case) guiding it is available.

Finally, in both experiments the majority of the participants positively perceived the usefulness of refactoring during TDD. The percentages are higher in Exp2 (92% and 83% with and without SonarLint) than in Exp1 (64% and 84%) and this may be due to participants' characteristics–*i.e.,* senior participants are more used to quality improvement tasks. Furthermore, we can notice that refactoring was perceived as more useful by the participants not using SonarLint in Exp1, while the opposite happened in Exp2. The participants in Exp2 have, from their course, a background about code review and continuous integration. That is, they were more used to take advantage from a tool support guiding refactoring.

3.5 Threats to Validity

To determine the threats that might affect the validity of our results, we followed Wohlin *et al.*'s guidelines [68]. Although we tried to mitigate/avoid as many threats to validity as possible, some of them are unavoidable. This is because mitigating/avoiding a kind of threat (*e.g.,* internal validity) might intensify/introduce another kind of threat (*e.g.,* external validity) [68]. Since we conducted the first study (comprising two experiments) investigating the use of a SAT when applying TDD, we preferred to reduce threats to internal validity (*i.e.,* making sure that the cause–effect relationships were correctly identified), rather than being in favor of external validity.

Threats to internal validity. A *selection* threat might have affected our results due to the volunteer participation—volunteers are generally more motivated than the whole population [68]. Another potential threat is *resentful demoralization*, which arises when a participant receives a less desirable treatment and then he/she does not behave as he/she normally would. This threat holds in both Exp1 and Exp2 and it is related to the adopted experimental design. Finally, we could not monitor the participants in Exp2 while executing the experimental task since they accomplished part of it outside a laboratory setting. This might poses a threat of *diffusion/treatment imitations*. However we asked the participants (in any experiment) to do their best alone and not to talk about the experiment with their classmates. Furthermore, any participant was informed about being rewarded (with bonus points) for participating in the experiment regardless of his/her performance. To some extent, this design choice discouraged the participants from exchanging information.

Threats to construct validity. Although we did not disclose our research goal to the participants, they might have tried to guess it and adapted their behavior based on their guess (threat of *hypotheses guessing*). As for a threat of *evaluation apprehension*, it should be mitigated as the participants knew that they would be rewarded for their participation regardless of their performance in the experiment. There might be a threat of *restricted generalizability across constructs*. Namely, while we found a positive effect of using SonarLint on software quality, there might be side effects (due to SonarLint) on unconsidered constructs. We mitigated this threat by checking that no significant difference emerged, between the treatments, in terms of: number of tackled user stories, number of tests, productivity, and functional quality (see our online material [47]).

Threats to conclusion validity. There might be a threat of *random heterogeneity of participants* despite we mitigated it through two countermeasures: *(i)* in each experiment, we involved students taking the same course, so having similar backgrounds and skills; and *(ii)* the participants in each experiment underwent a training to make them as more homogeneous as possible within the SonarLint and NoSonarLint groups. A threat of *reliability of treatment implementation* might have occurred in Exp1 and Exp2. For example, some participants might have followed TDD more strictly than others; however, this should equally affect both experimental groups. To mitigate this threat, we reminded the participants to apply TDD as more strictly as possible. Moreover, even if there are tools for checking the conformance to TDD (*e.g., Besouro* [8] and *WatchDog* [10]), they are obsolete or not yet available in the *Eclipse Marketplace*. As regards a threat of *reliability of measures*, we mitigate it by using metrics grounded on previous studies (*e.g.,* [13, 16, 37, 40, 66]). As for the analysis of refactorings, it might be affected by the tool performance, which is, nevertheless, quite good (99.6% precision and 94% recall [62]). At the same time, despite our guidelines and monitoring actions, we cannot exclude that the participants performed refactorings without making incremental

commits, making them invisible to the detection tool. A threat of *violated assumptions of statistical tests* might be present. Indeed, we did not check the normality assumption of LMMs [67] (in the IPD-S analyses) and Hedges' g (in the AD analysis). However, LMMs are robust to departures from normality [50], while there is yet no valid non-parametric alternative to Hedges' g (*i.e.,* while a number of non-parametric effect size measures are available, it is still unclear how to aggregate results from individual experiments to obtain an overall non-parametric effect size) [34]. The outcomes from IPD-S were always confirmed by AD so making us confident about the correctness of our statistical conclusions. Finally, we acknowledge a potential threat of *error rate* (*i.e.,* Type-I errors), since we performed multiple statistical tests, although on different hypotheses. We avoided using adjustment techniques as they increase the chance of making Type-II errors.

Threats to external validity. We involved Bachelor's and Master's students in our experiments, so potentially posing a threat of *interaction of selection and treatment* (*i.e.,* the results might not be generalized to professional developers). However, the use of students has the advantage that they have more homogeneous backgrounds and skills and are suitable to obtain initial empirical evidence [15, 27]. The used experimental object, MRA, might represent another threat to external validity: *interaction of setting and treatment*. However, MRA has been largely used in previous studies on TDD (*e.g.,* [21, 23, 32, 58]). Finally, we acknowledge that our results might not be generalized to a SAT different from SonarLint.

4 DISCUSSION

This section outlines implications for practitioners, educators, and researchers based on the main findings from our study.

Implications for practitioners. We found that using a SAT in the TDD process positively affects software quality, in terms of reduced number of smells, reduced technical debt, and increased code understandability. This can help mitigate a behavior found in previous studies, namely: developers frequently skip the Refactor phase when applying TDD [2, 48, 53]. Past work has also shown how developers leverage SATs during code reviews [3, 44, 55] and within continuous integration pipelines [70]. Our work highlights how, in the context of an agile development practice like TDD where some developers might not consider software quality as a first priority, the adoption of SATs can result useful already within developers' IDEs so helping them deliver high-quality software.

Implications for educators. The results in favor of SonarLint (*i.e.,* better software quality), but also those not in favor thereof (*e.g.,* TDD perceived as more difficult when using the SAT), suggest properly teaching agile practices (including TDD) along with practices for software quality improvement, including leveraging SATs, looking at metric indicators, and performing refactorings.

Implications for researchers. The post-experiment data highlight that both execution of the implementation task and application of TDD were perceived as more difficult when using SonarLint. A plausible explanation is that, while SonarLint helps identify refactoring opportunities by highlighting code smells directly in the IDE, an extra effort is needed to understand the rationale behind a given code smell and how to remove it. SonarLint provides a view where the developer can understand the rationale behind a code smell; however, the refactoring action is totally left up to the developer. In such scenario, an improvement of SonarLint could be achieved by integrating a refactoring recommender in the IDE [6, 33, 43, 59–61, 69]. Finally, when applying TDD, it can be desirable to have a supporting tool guiding both TDD and refactoring activities. In Section 3.3, we conjectured that, in Exp2, the participants using SonarLint mainly performed refactorings in the Green phase right after SonarLint highlighted code smells, hence making such refactorings not visible to RefactoringMiner. Previous work has shown that performing refactorings outside the Refactor phase is a common deviation from the TDD process [2]. We thus foster researchers to conceive a new generation of SATs specific for the TDD practice. Namely, SATs that, besides identifying refactoring opportunities while coding, check the conformance to TDD and warn practitioners when they deviate from the TDD process. This kind of SATs should highlight refactoring opportunities at the beginning of the Refactor phase.

5 RELATED WORK

In this section, we discuss and relate to our work research about TDD and the use of SATs during software development.

5.1 Studies on TDD

TDD is claimed to help deliver higher-quality software products, in terms of both *external* (*i.e.,* functional) and *internal* quality, while increasing developers' productivity [19]. Such claims have encouraged researchers to carry out an ample set of studies on TDD. Shull *et al.* [54] aggregated evidence from a selection of high-quality studies—in particular, experiments, pilot and industry studies—on the effectiveness of TDD with respect to internal and external quality, and productivity. In terms of internal quality (measured through code-complexity, code-density, or object-oriented structure metrics), TDD shows a non-consistent effect. External quality was addressed through metrics such as the percentage of tests passed or defect density, while productivity was assessed by measuring, for instance, development or maintenance effort. Different outcomes emerge from the various kinds of studies without any particular consistent negative or positive effect.

Over the years, other secondary studies have aggregated results on the claimed effects of TDD to provide joint conclusions. For example, Bissi *et al.* [11] carried out a Systematic Literature Review (SLR) on the effects of TDD on internal and external quality, and productivity, comparing TDD with *Test-Last Development* (*TLD*). The authors considered primary studies conducted from 1999 to 2014. The results point out that TDD leads to higher benefits in terms of internal (76% of the cases) and external quality (88%) while decreasing productivity (44%), as compared to TLD. Rafique and Misic [45] investigated how TDD impacts external quality and productivity through a meta-analysis conducted on a set of 27 studies. The authors reported that the use of TDD results in a small improvement in external quality of software, but the results on productivity are inconclusive. Munir *et al.* [41] in their SLR classified 41 primary studies, published from 2000 to 2011, into four categories based on high/low rigor and relevance. They found that in each category different conclusions can be drawn for internal and

external quality, and productivity. This implies that, when looking at these studies as a unique set, the results are inconclusive.

In spite of the several studies, either primary or secondary, on TDD, in most cases, their focus is on external quality and developers' productivity. Only few cases relate to internal quality, and when this occurs, it is limited to measuring code coverage [11]. In other words, compared to existing studies on TDD, we considered a wider range of metrics to measure (internal) software quality, ranging from code complexity to code readability metrics.

Finally, it is worth remarking that the assumption of the research presented in this paper is that (internal) software quality improvements can be triggered by indicators of SATs, able to identify software quality concerns, such as code smells, and suggestions to developers on how to improve software quality. As so, SATs can become an integrated part of the TDD process as a means for easing the identification of refactoring opportunities in the Refactor phase. Our results give credit to our assumption and seem to encourage the development of a new generation of SATs specific for TDD.

5.2 Use of SATs in Software Development

SATs can be beneficial for developers to early identify potential bugs, vulnerabilities, and performance issues, as well as to detect deviations from the project coding guidelines. Previous literature has investigated how developers use SATs during software development [63, 64]. Specifically, Johnson *et al.* [29], by interviewing 20 developers shed light on the reasons why developers underuse SATs despite their benefits—*i.e.,* false alarms and the poor understandability of the output being generated. Spacco *et al.* [57] found that developers tend to focus more on high and medium priority warnings, while low priority warnings are less removed over time.

As for the relationship between fault occurrences and SATs warnings, Zheng *et al.* [71] showed that SATs can be used for the early discovery of security vulnerabilities. Couto *et al.* [18], instead, highlighted that warnings do not contribute to locating pieces of code affected by software defects; however, they can be used as early indicators for future defects. The latter is confirmed by a more recent study by Lenarduzzi *et al.* [36], studying the fault-proneness of the SonarQube rules, highlighting that only 25 out of 202 rules being available for Java projects have a relatively low fault-proneness.

Beller *et al.* [9] conducted a large-scale evaluation on the usage of SATs in open-source projects pointing out that their usage is not widespread, but also their adoption is highly dependent on the programming language used by the projects. Panichella *et al.* [44] investigated whether it is possible to use SATs in the context of code reviews. Their results show that the density of warnings only slightly varies after each review; however, there are warnings—*e.g.,* dealing with imports, regular expressions, and type resolution—for which the removal percentage is very high (*i.e.,* above 50% and sometimes up to 100%). Zampetti *et al.* [70], looked into how developers use SATs in continuous integration and delivery pipelines, highlighting that the adherence to coding standards is the main root cause for build failures due to SATs, and developers quickly fix the warnings, rather than simply hiding the problem (*i.e.,* by disabling the warning in the SAT configuration). Finally, a study by Vassallo *et al.* [63], by mining the history of open source projects and surveying industrial and open source contributors, examined how developers use SATs in different development contexts, as well as how they integrate SATs in their workflows. Their results indicate that: *(i)* the type of warning categories being considered is highly dependent on the development context in which SATs are used; *(ii)* the warnings selection and prioritization are mainly dictated by factors like team policies and composition; and *(iii)* even if 66% of projects have guidelines aimed to understand how SATs have to be used, only 37% of them enforce their usage.

Violations of coding rules can be seen as technical debt items identified by SATs such as SonarQube. Baldassarre *et al.* [4] conducted a case study where 81 junior developers were asked to fix technical debt items in 21 open-source Java projects, revealing that SonarQube's remediation time is in most cases overestimated.

Our study advances the state of the art since we analyze whether the use of a SAT influences the application of TDD in terms of software quality, considering that developers often skip the Refactor phase [2, 48, 49], potentially leading to software of poor quality.

6 CONCLUSION AND FUTURE WORK

In this paper, we present the results of two experiments aimed to investigate the benefits of leveraging a Static Analysis Tool (SAT) on software quality, when developers apply an agile software development practice like Test-Driven Development (TDD). Specifically, the participants involved in the two experiments accomplished an implementation task by applying TDD, with or without a SAT highlighting the presence of code smells in their source code.

The overall results of the experiments suggest that the use of a SAT not only helps, as expected, to significantly decrease the number of code smells, but also reduces the estimated technical debt and improves the measured code understandability. Based on the obtained results, we delineated possible implications from the perspectives of researchers, practitioners, and educators. We foster researchers to define a new generation of SATs that, besides highlighting refactoring opportunities in the IDE (Integrated Development Environment), check the conformance to TDD and warn developers when they deviate from the TDD process. These new SATs could help TDD practitioners to identify refactoring opportunities and avoid that they skip the Refactor phase. After all, we show that TDD practitioners can benefit from the use of SATs in terms of improvements in software quality. Our positive results (*i.e.,* the use of a SAT leads to better software quality) and negative ones (*e.g.,* TDD is perceived as more difficult when using a SAT) suggest educators properly teach TDD along with practices for software quality improvement, including leveraging SATs, looking at metric indicators, and performing refactorings whenever necessary.

Our results can inspire further and different studies (*e.g.,* case studies) to experiment with the use of other kinds of SATs. Also, it can be interesting to compare the benefits introduced by SATs at different development stages. Finally, despite we gather evidence that the TDD practice can take advantage of the use of a SAT, we foster replications of our experiments, especially by involving professional developers and the software industry. Our experiments have the merit to justify such replications—it is easier to recruit professional developers and involve the software industry when initial evidence is available. To ease the replicability of the experiments, our online material includes a replication package.

REFERENCES

[1] Maurício Aniche. 2015. *Java code metrics calculator (CK).* Available in https://github.com/mauricioaniche/ck/.

[2] Mauricio Finavaro Aniche and Marco Aurélio Gerosa. 2010. Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 469–478. https://doi.org/10.1109/ICSTW.2010.16

[3] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*. 931–940. https://doi.org/10.1109/ICSE.2013.6606642

[4] Maria Teresa Baldassarre, Valentina Lenarduzzi, Simone Romano, and Nyyti Saarimäki. 2020. On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. *Information and Software Technology* 128 (2020), 106377.

[5] Victor Basili, Gianluigi Caldiera, and H. Dieter Rombach. 2002. *Goal Question Metric (GQM) Approach.* John Wiley & Sons, Ltd, 528–532. https://doi.org/10.1002/0471028959.sof142

[6] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Trans. Software Eng.* 40, 7 (2014), 671–694. https://doi.org/10.1109/TSE.2013.60

[7] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley.

[8] Karin Becker, Bruno de Souza Costa Pedroso, Marcelo Soares Pimenta, and Ricardo Pezzuol Jacobi. 2015. Besouro: A framework for exploring compliance rules in automatic TDD behavior assessment. *Information and Software Technology* 57 (2015), 494–508.

[9] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[10] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering* 45, 3 (2019), 261–284. https://doi.org/10.1109/TSE.2017.2776152

[11] Wilson Bissi, Adolfo Gustavo Serra Seca Neto, and Maria Cláudia Figueiredo Pereira Emer. 2016. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Inf. Softw. Technol.* 74 (2016), 45–54. https://doi.org/10.1016/j.infsof.2016.02.004

[12] Michael Borenstein, Larry Hedges, Julian Higgins, and Hannah Rothstein. 2009. *An Introduction to Meta-Analysis.* John Wiley & Sons.

[13] Raymond P. L. Buse and Westley Weimer. 2010. Learning a Metric for Code Readability. *IEEE Trans. Software Eng.* 36, 4 (2010), 546–558. https://doi.org/10.1109/TSE.2009.70

[14] G. Ann Campbell. 2018. Cognitive Complexity: An Overview and Evaluation. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) *(TechDebt '18)*. Association for Computing Machinery, 57–58. https://doi.org/10.1145/3194164.3194186

[15] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. 2003. Issues in Using Students in Empirical Studies in Software Engineering Education. In *Proceedings of International Symposium on Software Metrics.* IEEE, 239–249.

[16] Shyam R. Chidamber and Chris. F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895

[17] Jacob Cohen. 1992. A power primer. *Psychological bulletin* 112 (1992), 155–9. Issue 1.

[18] César Couto, João Eduardo Montandon, Christofer Silva, and Marco Tulio Valente. 2013. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Softw. Qual. J.* 21, 2 (2013), 241–257. https://doi.org/10.1007/s11219-011-9172-5

[19] Hakan Erdogmus, Grigori Melnik, and Ron Jeffries. 2010. Test-Driven Development. In *Encyclopedia of Software Engineering.* Taylor & Francis, 1211–1229.

[20] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (1st ed.). Addison-Wesley.

[21] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering* 43, 7 (2017), 597–614.

[22] Davide Fucci, Giuseppe Scanniello, Simone Romano, and Natalia Juristo. 2018. Need for Sleep: the Impact of a Night of Sleep Deprivation on Novice Developers' Performance. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2834900

[23] Davide Fucci, Giuseppe Scanniello, Simone Romano, Martin J. Shepperd, Boyce Sigweni, Fernando Uyaguari Uyaguari, Burak Turhan, Natalia Juristo, and Markku Oivo. 2016. An External Replication on the Effects of Test-driven Development Using a Multi-site Blind Analysis Approach. In *International Symposium on Empirical Software Engineering and Measurement.* 3:1–3:10.

[24] Mohammad Ghafari, Timm Gross, Davide Fucci, and Michael Felderer. 2020. Why Research on Test-Driven Development is Inconclusive?. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Association for Computing Machinery, New York, NY, USA, Article 25, 10 pages. https://doi.org/10.1145/3382494.3410687

[25] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Earlbaum Associates.

[26] Julian P.T. Higgins, James Thomas, Jacqueline Chandler, Miranda Cumpston, Tianjing Li, Matthew J. Page, and Vivian A. Welch. 2019. *Cochrane Handbook for Systematic Reviews of Interventions.* John Wiley & Sons.

[27] Martin Höst, Björn Regnell, and Claes Wohlin. 2000. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empir. Softw. Eng.* 5, 3 (2000), 201–214.

[28] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. 2008. Reporting Experiments in Software Engineering. In *In Guide to Advanced Empirical Software Engineering*. Springer, 201–228.

[29] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[30] Natalia Juristo and Ana M. Moreno. 2001. *Basics of Software Engineering Experimentation.* Kluwer Academic Publishers.

[31] Itir Karac and Burak Turhan. 2018. What Do We (Really) Know about Test-Driven Development? *IEEE Softw.* 35, 4 (2018), 81–85.

[32] Itir Karac, Burak Turhan, and Natalia Juristo. 2021. A Controlled Experiment with Novice Developers on the Impact of Task Description Granularity on Software Quality in Test-Driven Development. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1315–1330. https://doi.org/10.1109/TSE.2019.2920377

[33] Marouane Kessentini, Troh Josselin Dea, and Ali Ouni. 2017. A Context-Based Refactoring Recommendation Approach Using Simulated Annealing: Two Industrial Case Studies. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 1303–1310. https://doi.org/10.1145/3071178.3071334

[34] Barbara Kitchenham, Lech Madeyski, and Pearl Brereton. 2020. Meta-analysis for families of experiments in software engineering: a systematic review and reproducibility and validity assessment. *Empirical Software Engineering* 25, 1 (Jan. 2020), 353–401. https://doi.org/10.1007/s10664-019-09747-0

[35] Barbara Ann Kitchenham, Lech Madeyski, Giuseppe Scanniello, and Carmine Gravino. 2021. The Importance of the Correlation in Crossover Experiments. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3070480

[36] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. 2020. Are sonarqube rules inducing bugs?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 501–511.

[37] Jean-Louis Letouzey. 2012. The SQALE method for evaluating Technical Debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*. 31–36. https://doi.org/10.1109/MTD.2012.6225997

[38] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 209–219. https://doi.org/10.1109/ICPC.2019.00040

[39] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). Prentice Hall.

[40] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. 2020. An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3382494.3410636

[41] Hussan Munir, Misagh Moayyed, and Kai Petersen. 2014. Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology* 56, 4 (2014), 375–394. https://doi.org/10.1016/j.infsof.2014.01.002

[42] Bram Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement.* Pinter Publishers.

[43] Fabio Palomba, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Andrian Marcus, Denys Poshyvanyk, and Andrea De Lucia. 2015. Extract Package Refactoring in ARIES. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. 669–672. https://doi.org/10.1109/ICSE.2015.219

[44] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews?. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 161–170.

[45] Yahya Rafique and Vojislav B. Mišić. 2013. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE Transactions on Software Engineering* 39, 6 (2013), 835–856. https://doi.org/10.1109/TSE.2012.28

[46] Jeanine Romano, Jeffrey D. Kromrey, Jesse. Coraggio, Jeff Skowronek, and Linda Devine. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?. In *Annual Meeting of the Florida Association of Institutional Research, February*. 1–3.

[47] Simone Romano. 2022. Do Static Analysis Tools Affect Software Quality when Using Test-driven Development? Online Material. (2022). https://doi.org/10.6084/m9.figshare.20197832.v1

[48] Simone Romano, Davide Fucci, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. 2017. Findings from a multi-method study on test-driven development. *Inf. Softw. Technol.* 89 (2017), 64–77.

[49] Simone Romano, Giuseppe Scanniello, Carlo Sartiani, and Michele Risi. 2016. A Graph-based Approach to Detect Unreachable Methods in Java Software. In *Proceedings of Symposium on Applied Computing*. ACM, 1538–1541.

[50] Adrian Santos, Sira Vegas, Markku Oivo, and Natalia Juristo. 2019. A Procedure and Guidelines for Analyzing Groups of Software Engineering Replications. *IEEE Trans. Softw. Eng.* (2019), 1–1.

[51] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: how far are we?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 417–427. https://doi.org/10.1109/ASE.2017.8115654

[52] Giuseppe Scanniello, Carmine Gravino, Marcela Genero, José A. Cruz-Lemus, Genoveffa Tortora, Michele Risi, and Gabriella Dodero. 2018. Do software models based on the UML aid in source-code comprehensibility? Aggregating evidence from 12 controlled experiments. *Empir. Softw. Eng.* 23, 5 (2018), 2695–2733.

[53] Giuseppe Scanniello, Simone Romano, Davide Fucci, Burak Turhan, and Natalia Juristo. 2016. Students' and Professionals' Perceptions of Test-Driven Development: A Focus Group Study. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. Association for Computing Machinery, New York, NY, USA, 1422–1427. https://doi.org/10.1145/2851613.2851778

[54] Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, and Hakan Erdogmus. 2010. What Do We Know about Test-Driven Development? *IEEE Software* 27, 6 (2010), 16–19. https://doi.org/10.1109/MS.2010.152

[55] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 101–105. https://doi.org/10.1109/VLHCC.2017.8103456

[56] SonarSource SA. 2022. SonarLint. https://www.sonarlint.org. (Last access: 02/05/2022).

[57] Jaime Spacco, David Hovemeyer, and William Pugh. 2006. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*. 133–136.

[58] Ayse Tosun, Oscar Dieste, Davide Fucci, Sira Vegas, Burak Turhan, Hakan Erdogmus, Adrian Santos, Markku Oivo, Kimmo Toro, Janne Jarvinen, and Natalia Juristo. 2017. An industry experiment on the effects of test-driven development on external quality and productivity. *Empirical Software Engineering* 22, 6 (2017), 2763–2805.

[59] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Extract Method Refactoring Opportunities. In *13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009*. 119–128. https://doi.org/10.1109/CSMR.2009.23

[60] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Trans. Software Eng.* 35, 3 (2009), 347–367. https://doi.org/10.1109/TSE.2009.1

[61] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2010. Identification of refactoring opportunities introducing polymorphism. *J. Syst. Softw.* 83, 3 (2010), 391–404. https://doi.org/10.1016/j.jss.2009.09.017

[62] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 03 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[63] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* 25, 2 (2020), 1419–1457. https://doi.org/10.1007/s10664-019-09750-5

[64] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 38–49. https://doi.org/10.1109/SANER.2018.8330195

[65] Sira Vegas, Cecilia Apa, and Natalia Juristo. 2016. Crossover Designs in Software Engineering Experiments: Benefits and Perils. *IEEE Transactions on Software Engineering* 42, 2 (2016), 120–135. https://doi.org/10.1109/TSE.2015.2467378

[66] William C. Wake. 2003. *Refactoring Workbook* (1st ed.). Addison-Wesley.

[67] Anne Whitehead. 2003. *Meta-Analysis Of Controlled Clinical Trials*. John Wiley & Sons.

[68] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer.

[69] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017. GEMS: An Extract Method Refactoring Recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 24–34. https://doi.org/10.1109/ISSRE.2017.35

[70] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 334–344. https://doi.org/10.1109/MSR.2017.2

[71] Jiang Zheng, Laurie A. Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering* 32, 4 (2006), 240–253. https://doi.org/10.1109/TSE.2006.38