Investigating Severity Thresholds for Test Smells

Davide Spadini
d.spadini@sig.eu
Software Improvement Group &
Delft University of Technology
Amsterdam, The Netherlands

Martin Schvarcbacher
martin.schvarcbacher@student.uva.nl
Software Improvement Group
Amsterdam, The Netherlands

Ana-Maria Oprescu
A.M.Oprescu@uva.nl
University of Amsterdam
Amsterdam, The Netherlands

Magiel Bruntink
m.bruntink@sig.eu
Software Improvement Group
Amsterdam, The Netherlands

Alberto Bacchelli
bacchelli@ifi.uzh.ch
University of Zurich
Zurich, Switzerland

ABSTRACT

Test smells are poor design decisions implemented in test code, which can have an impact on the effectiveness and maintainability of unit tests. Even though test smell detection tools exist, how to rank the severity of the detected smells is an open research topic. In this work, we aim at investigating the severity rating for four test smells and investigate their perceived impact on test suite maintainability by the developers. To accomplish this, we first analyzed some 1,500 open-source projects to elicit severity thresholds for commonly found test smells. Then, we conducted a study with developers to evaluate our thresholds. We found that (1) current detection rules for certain test smells are considered as too strict by the developers and (2) our newly defined severity thresholds are in line with the participants' perception of how test smells have an impact on the maintainability of a test suite. Preprint [https://doi.org/10.5281/zenodo.3744281], data and material [https://doi.org/10.5281/zenodo.3611111].

KEYWORDS

Test Smells, Software Testing, Empirical Software Engineering

ACM Reference Format:
Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. 2020. Investigating Severity Thresholds for Test Smells. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387453

1 INTRODUCTION

Violations of design principles (*a.k.a.*, code smells) are not restricted to production code, but are also found in (unit) test code [12, 19, 31]. Such *test smells* can lead to harder to maintain tests [6, 7, 29], just as (production) code smells can increase maintenance effort [27].

Developers tend to focus on production code quality, while test code quality is often not prioritized [29]; moreover, once test smells are introduced, they are hardly ever removed through refactoring [30]. One could argue that the concept of test code quality and test smells in particular is in need of further investigation. For example, previous research has reported that developers do not always perceive test smells as problematic [30], but the actual reason is unclear. One reasonable explanation is that current test smell detection tools lack *severity thresholds*, which could make their indications more actionable. Indeed, Alves *et al.* showed the importance of determining severity thresholds for (production) code smells to "adequately support subsequent decision-making" [2], by successfully using their defined thresholds for software analysis, benchmarking, and certification in industry [2]. In this study, we investigate severity thresholds for test smells.

Particularly, our aim is to:

- Calibrate detection thresholds such that severity levels can be assigned to a test smell instance, allowing developers to focus on the higher severity smells.
- Improve the accessibility of automatic test smell detection by integrating into existing developer tooling.

In their work, Alves *et al.* defined a method to calibrate thresholds for code quality metrics, which they named as 'benchmark based threshold derivation methodology' [2]. Their approach consisted of collecting data from existing software systems and using the distributions of metric values to find appropriate thresholds [2, 4]. In our investigation, we take a similar approach. We collect the (unit) test code of 1,489 Java projects from the Apache and Eclipse ecosystems and apply the open-source test smells detection tool tsDetect [23]; then, we use these systems as a benchmark to derive the values for three severity thresholds: 'Medium', 'High', 'Very high'. Following this approach, we found that four of nine test smells we considered should have higher thresholds than what previously reported in literature [7, 22, 29].

Subsequently, we move to our second goal: We integrate test smell detection provided by tsDetect into a prototype (back-end) extension of BetterCodeHub [1] (BCH), a web-based code quality analysis tool provided by SIG. We engaged the existing users of BCH to interact with the new test smell prototype and solicited their feedback on instances of test smells within their own code bases. A total of 31 developers, across 47 diverse projects, answered,

[1] https://bettercodehub.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387453

providing data points on 301 detected test smells. These responses allowed us to evaluate the developers' perceptions of our newly proposed thresholds. According to the developers, Empty Test, Sleepy Test, and Mystery Guest have the highest priority as refactoring candidates, while Empty Test, Ignored Test, and Conditional Test Logic are considered the smells with the higher impact on code maintainability. Furthermore, the ratings submitted by the users are aligned with our thresholds, with a statistically significant difference and a strong Spearman's coefficient, suggesting that the newly defined thresholds can be used to prioritize test smells instances.

Our study makes the following contributions:

(1) Calibrated severity thresholds for test smell detectors
(2) A mechanism for rating the severity of certain test smells, allowing tools to classify test smells into distinct categories based on their severity. This enables developers to only focus on the most critical test smells.
(3) An integration of an automatic test smell detector within a GitHub-based code quality tool (BCH).
(4) An evaluation of developers' perceptions of test smells within their own code bases, by integrating our new thresholds into a GitHub-based code quality tool (BCH), thus validating our test smell severity levels.

2 RELATED WORK

2.1 Test Smell Detection Tools

Van Rompaey *et al.* created a metrics-based test smell detection tool for Java to detect General Fixtures and Eager Tests [32, 33]. Later Breugelmans and van Rompaey developed the TestQ tool, which works with C, C++ and Java test code to detect the following test smells: Assertion Roulette, Eager Test, Empty Test, For Testers Only, General Fixture, Indented Test (the equivalent of Conditional Test Logic), Indirect Test, Mystery Guest, Sensitive Equality and Verbose Test [8].

Greiler *et al.* focused on identifying common problems with test fixtures. They implemented a tool called TestHound, which works on JVM bytecode level and can detect the following test smells: General Fixture, Test Maverick, Dead Field, Lack of Cohesion of Test Methods, Obscure In-Line Setup and Vague Header. In this tool, they showed the code impacted by test smells to the developers along with tips for how to refactor the test code [14]. This setup is comparable to the one we implemented in BCH, as we also present code instances impacted by test smells. They concluded that having a tool to point out the problems with the test suite can help the developers with refactoring.

Palomba *et al.* developed TASTE (Textual AnalySis for Test smEll detection), which can detect General Fixtures, Eager Tests, and Lack of Cohesion of Methods using Information Retrieval techniques, thus bypassing the need to fully parse the test code. Their tool shows a better precision and recall than the AST-based tools TestQ and TestHound [22].

Satter *et al.* [25] created a tool for the detection of dead fields in Java unit tests. They report a better detection accuracy than TestHound.

Bavota *et al.* studied the diffusion of test smells in open-source and industrial projects [6]. To help with their research, they created a test smell detection tool for Resource Optimism, Indirect Testing, Test Run War, Mystery Guest, General Fixture, Eager Test, Lazy Test, Assertion Roulette, For Testers Only, Test Code Duplication and Sensitive Equality.

Zhang *et al.* focused on dependencies between tests by empirically investigating issue tracking systems. They developed a tool which identifies dependencies between tests on a test suite level, not on individual test case level). Their approach requires executing the tests (dynamic analysis) [34].

Gambi *et al.* also looked into test dependency detection and developed the tool PraDeT, which also requires running the tests to find dependencies [13].

2.2 How Developers Perceive Test Smells

Tufano *et al.* asked 19 developers from various open-source projects to look at test code samples which contained instances of test smells. In the majority of the presented cases (82%), the developers did not recognize any problems with the test code. The code presented was created or maintained by the interviewed developers and thus the developers saw or created the presented code before the interview session. The authors highlight the need for having automated tools that can detect test smells and present them to the developers. They have also found out that the majority of test smells are created during the initial test development and are not removed in subsequent refactoring [30].

Palomba *et al.* investigated the developer's perception of code smells, using both the original authors and independent developers. They asked the developers to identify the design problems (code smells) and, if found, give them a severity rating. The severity ratings were applied to the whole code smell category, and not to the specific code smells instances. Furthermore, the research focused only on production code smells and not test smells. They have then split the observed code smells into 3 categories: Generally not Perceived as Design or Implementation Problems, Generally Perceived and Identified by Respondents, and Perception may Vary [21].

Both of these studies differ from ours in execution, as they presented developers test code sections without further context, and outside of the developer's usual work environments. In contrast, our study asks developers to work within the normal work-flow of their code quality tool (BCH) and within the context of their own project.

Kummer studied whether developers recognize test smells using a sample of 20 developers. The author concludes that test smells can be refactored by the developers without them knowing that they are specific instances of test smells and suggests that automated test smell detection tools could help the developers further justify their removal [15].

2.3 Test Smells and Code Quality

Spadini *et al.* studied the relationship between the presence of test smells and software change and defect proneness. They analyzed multiple releases of ten software products and their test cases, investigating the following test smells: Mystery Guest, Resource Optimism, Eager Test, Assertion Roulette, Indirect Testing and Sensitive Equality. Between each release, they looked at how the production code changed and how many fixes were reported by the accompanying issue tracking systems and Git commits. Their

main findings were that "tests affected by test smells are associated with higher change- and defect-proneness than tests not affected by smells" [29]. In our research, we investigate a wider variety of test smells and use an existing code quality model to compare against, however we do not investigate change- and defect proneness.

3 METHODOLOGY

Our overall research goal is to investigate severity thresholds for test smells. The most common test smell detection tools work on a binary scale of whether a given code is impacted by a test smell or not without any additional data, thus ignoring commonly encountered situations in software development.

In our first research question, we seek to define new thresholds for test smells, based on the 'benchmark based threshold derivation methodology' [2] considering a large number of software systems.

> **RQ$_1$.** *How can test smells be given a severity rating?*

In our second research question, we aim to challenge our newly defined thresholds with users, therefore we ask:

> **RQ$_2$.** *What is the perception of developers on test smells in their codebase?*

3.1 Test Smells Tool Selection

We researched the available test smell detection tools, detailed in Section 2.1. Given that most of the better validated tools only support Java for the widest variety of test smells, we decided to focus on test smell research in Java. We selected the open-source tool tsDetect [23] for finding test smells in the codebase. The tool works with Java JUnit projects and has support for JUnit 4 annotations, which can be extended to support JUnit 5, and has precision and recall above 85% [23]. It also has a published accuracy rating along with manually classified test smell data [23], providing a test suite for the tool extension and testing. The main advantages of tsDetect are that it is open-source, developed recently in 2018, uses AST-based detection of test smells, and supports adding new test smells and detection rules. Compared to other tools, the support for JUnit along with using only AST information instead of text search for pattern violations results in improved accuracy.

3.2 Test Smells Selection

tsDetect can detect multiple test smells. However, for this study we restricted our analysis to the test smells depicted in Table 1. We select these test smells because they do not require viewing the full test source code to understand whether they are smelly or not. Certain test smells, such as Lazy Test (multiple test methods invoking the same method of the production object), require viewing the entire test class code to evaluate, while in this research we are focused on detecting test smells at method level. Based on testing done on a selected dataset, tsDetect is highly reliable at detecting instances of these test smells [23]. Previous studies [7, 24] report the distributions of the smells we analyzed along with others we did not select. The F-Score for tsDetect on the selected subset of test smells is between 87% and 99% [23].

3.3 Preliminary Study

Before running the study in large scale, we perform a preliminary study within SIG. In this phase, we test the integration of tsDetect (without any modification) in BetterCodeHub.

During the pilots, the developers were asked to answer three questions: (1) whether the test smell instance is valid in the project context, (2) whether they classify the smell as "refactoring candidate" or as something which will take too much time and effort to fix (technical debt), and (3) rate from 1 to 5 the importance of the test smell instance on the project's maintainability. After completing the pilot, the developers were interviewed about their experience and asked for additional feedback, so that we could implement them before running the actual study. Four developers of SIG participated in the pilot, and each of them was asked to evaluate parts of the codebase written in Java they were actively working on as part of their daily job while using our prototype for the evaluation.

The main result of this preliminary study is that many of the test smell instances were rated as false positive or no-fix. The main reason is that in many of these instances developers did not see a design issue. For example, for the "Conditional Test Logic" smell developers complained that having only one branch (*e.g.*, *if-else*, or a *for*) should not be considered as a high priority refactoring. Similarly, developers did not agree on how "Eager Test" is calculated: currently, a method suffers from this smell if it contains more than one production call. However, developers agreed that almost all the tests have at least two production calls, hence resulting in many test methods rated as Eager Tests even though they are not.

On the other hand, in some cases developers acknowledged the problem and were willing to refactor the test method. However, since there is no distinction between having one production call or 10, these cases were mixed together with the false positives.

The main takeaway of this preliminary study is that test smells presented to the developers should be prioritized. The prioritization could help them focus only on the most severe test smell instances first, before moving onto fixing the less severe ones. The current implementation of test smell detection does not take into account additional information about the test smells, such as what percentage of the test code it impacts or how severe it is. Presenting the developers with test smells they perceive to be of low severity could lead them being more likely to ignore them and also trust the tool less. For these reasons, we need to determine a way to prioritize test smells, by means of new severity thresholds.

3.4 RQ1: Defining Severity Thresholds

With RQ1 we aim at creating new severity ratings for each test smell. Hence, we first define metrics for each test smell, based on their definitions and characteristics as proposed by van Deursen *et al.* [31] and Meszaros [19]. The metrics are shown in Table 2. Here the metrics denote the value which will be collected for each test method. Due to the binary nature of Empty Test and Ignored Test, we excluded them from the calibration metrics, and all instances of these smells are classified as the highest severity.

Once the metrics are defined, we need to give them a severity rating. For this, we used the Benchmark-based threshold derivation proposed by Alves *et al.* [2] methodology, which follows three core principles [11], which states that the method should (1) be

Table 1: Subject test smells

Test smell	Description	Problem
'Mystery Guest'	A test that uses external resources (e.g., file containing test data) [31]	Lack of information makes it hard to understand. Moreover, using external resources introduces hidden dependencies: if someone deletes such a resource, tests start failing.
'Resource Optimism'	A test that makes optimistic assumptions about the state/existence of external resources [31]	It can cause non-deterministic behavior in test outcomes. The situation where tests run fine at one time and fail miserably the other time.
'Eager Test'	A test that tries to verify too many functionalities, which can lead to difficulty in understanding the test code [31]	It is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.
'Assertion Roulette'	A test that has multiple assertion statements that do not provide any description of why they failed [31]	When the test fails, investigating the reason can be problematic as the assertion statement might not provide the reason why the test could have failed.
'Conditional Test Logic'	A test that has control flow statements inside a test [19]	tests can have multiple branch points and greater care must be taken when analyzing whether the test is correct.
'General Fixture'	It occurs when the test setup method creates fixtures (class fields used by the test cases) and a portion of the tests use only a subset of the fixtures [31]	The presence of this smell can result in longer execution of tests, as tests that do not use all of the fixtures still need to wait for the creation of all test fixtures.
'Empty Test'	It occurs when a test method does not contain executable statements	An empty test can be considered problematic and more dangerous than not having a test case at all since JUnit will indicate that the test passes even if there are no executable statements present in the method body.
'Magic Number Test'	A test that contains so-called "magic numbers", which are numbers used in assertions without any explanation where they come from and their value may not be immediately clear from just looking at the test code [17, 19]	The magic numbers should be replaced by a named constant, where the name describes where the value comes from or what it represents.
'Sleepy Test'	A test which contains thread suspension calls [19]	The use of this method call introduces additional delay to the test execution.
'Verbose Test'	A test that is too long and hard to understand [19]	Too long tests prevent them from being used as documentation and they are harder to maintain due to their complexity.
'Ignore Test'	A test method or class that contains the @Ignore annotation	ignored test methods result in overhead since they add unnecessary overhead with regards to compilation time, and increases code complexity and comprehension.

Table 2: Metrics used for threshold derivation, by test smell

Test Smell	Metric
Assertion Roulette	# assertions without description
Conditional Test Logic	# conditional statements
Eager Test	# production method calls
General Fixture	# unused fixtures
Magic Number Test	# magic numbers
Mystery Guest	# external files used
Resource Optimism	# files not checked for existence
Sleepy Test	# thread suspend calls
Verbose Test	# statements

driven by measurement data from a representative set of systems (data-driven), rather than expert opinion, (2) respect the statistical properties of the metric (*e.g.*, the metric scale and distribution), (3) be resilient against outliers in metric values and system size (robust), (4) be repeatable, transparent, and straightforward to carry out (pragmatic). We chose this technique as it (i) does not assume the normality of the metric values distribution, (ii) uses a weight function (LOC), which emphasizes the metric variability, (iii) separates the thresholds into different risk categories. Furthermore, this state-of-art benchmarking technique has been used in many previous studies that needed to calculate thresholds for new metrics [1, 3, 5, 11, 29].

The Benchmark-based threshold derivation enables us to define severity levels based on the representation of occurrences in the benchmark dataset.

Dataset. To apply this technique and derive the thresholds, we need a large and representative enough dataset. For this purpose, we selected *all* the projects from the Apache Software Foundation[2] and the Eclipse Foundation[3], which contained Java code. A total of 1,489 projects were selected. We use these projects because (1) the source code is publicly available, and (2) systems have different sizes and scopes. This will serve as our dataset. The dataset has a total of 25,356,827 LOC (project average: 31,617, median: 6,650).

[2]https://www.apache.org/
[3]https://www.eclipse.org/

Thresholds calculation. After the first step of selecting the projects, we follow the Benchmark-based threshold derivation methodology [2], consisting of the following 6 steps:

(1) **Metrics extraction**: for each test method in the systems, we extract test smell metric information (see Table 2) and LOC to be used as weight.
(2) **Weight ratio calculation**: for each method, we compute the weight percentage within its system, *i.e.*, we divide the method weight by the sum of all weights of the same system.
(3) **Entity aggregation**: we aggregate the weights of all methods
(4) **System aggregation**: we normalize the weights for the number of systems and then aggregate the weight for all systems.
(5) **Weight ratio aggregation**: we order the metric values in ascending order.
(6) **Thresholds Derivation**: we find the 70-80-90 percentiles to determine the thresholds.

After repeating this procedure for each test smell, we obtain 3 thresholds per smell for which we have defined a metric and weight. For instance, to represent 90% of the overall code for the Eager Test metric, the derived threshold is 39. In other words, we can say that 90% of the test methods (weighted by LOC and system size) have less than 39 production calls. This threshold is meaningful, since not only it means that it represents 90% of the code of a benchmark of systems, but it also can be used to identify 10% of the worst code [2]. To notice that the old threshold for this specific smell was the value one. We chose the percentile ranges of 70, 80, and 90 for each test smell, as this represents the increasingly worse portion of the codebase by volume [2, 4, 29] and thus represents the severity.

The thresholds allow the classification of test smells in a codebase into 4 distinct categories: test smell not found or below threshold (percentile intervals [0,70)), medium severity test smell for long term refactoring (percentile intervals [70,80)), high severity test smell for short term refactoring (percentile intervals [80,90)) and very high severity test smell for immediate refactoring (percentile intervals [90,100]).

3.5 RQ2: Developers' perceptions

To study the perception of developers on test smells found in their codebase, based on our thresholds, we modified BCH to present found test smells to the developers who could then provide their insights. By using the developer's repositories, we avoid the problem of developers seeing the code without knowing the project's context. Furthermore, we perform an analysis of test smells that were not considered for removal by the developers.

User Interface Design. Following the results of the new thresholds derived in RQ1, we modify BCH to integrate the test smells detector with the new thresholds. The front-end was also modified to show the analysis results along with our survey to gather developer feedback for each found test smell. This allows the developers who use BCH to access the results as part of their regular workflow, whether as a one-off analysis or continuous monitoring during pull requests or on a per commit level. Figure 1 shows an example of how the result was presented to the user.

Clicking on the test smell instance brings the users to the screen in Figure 2, where they can see the source code of the problematic code along with the survey. The users also had the ability to submit a GitHub issue to their repository with a description of the test smell and how it could be removed. The top displays a short explanation of what the test smell is and why it is harmful with a link to a more detailed page containing examples and explanations.

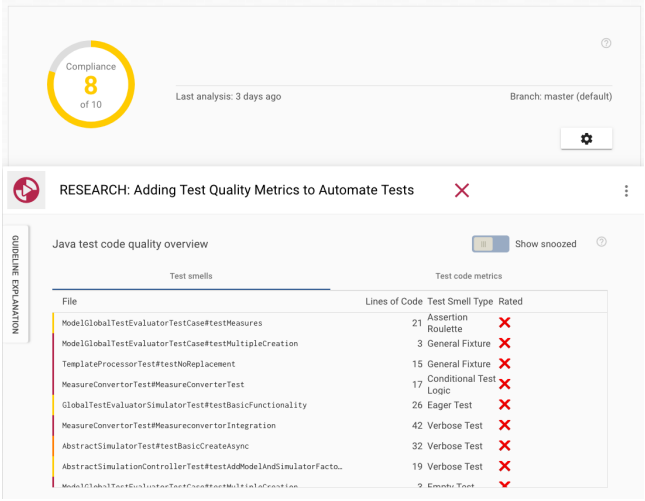


Figure 1: BCH view of test smell guidelines and violations.

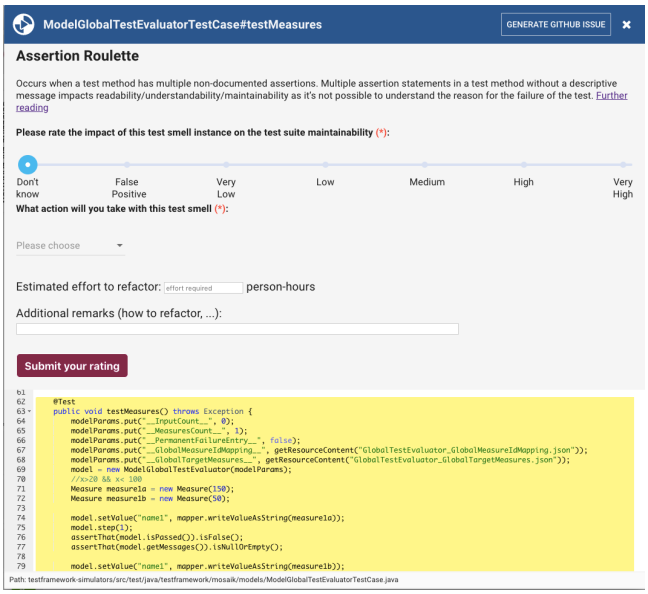


Figure 2: BCH details on a found test smell with the corresponding survey form.

Questions' Design. The questions asked to users about each found test smell are the following:

SQ1 Please rate the impact of this test smell instance on the test suite maintainability. *Options:* Don't know, False Positive, Very Low, Low, Medium, High, Very High.

SQ2 What action will you take with this test smell? *Options:* Mark for refactoring as soon as possible; Mark as technical debt for later refactoring; Ignore as won't fix / false positive.

SQ3 Estimated effort to refactor in person-hours. *Options:* numeric value (optional)

SQ4 Additional remarks. *Options:* free text (optional)

To avoid duplication, the questions (Figure 2) are only displayed if the user has not submitted a rating for the specific smell instance.

Attracting Participants. We sent an email to all the BCH users ($N \approx 13,000$) to invite them to test these new changes on a dedicated research server. The email also contained a brief text and video guideline on how to use these new features. A total of 31 users responded, providing feedback on more than 300 test smell instances, which have been all analyzed.

4 RESULTS

In this section we present our results by research question.

4.1 RQ1: Severity Thresholds

We analyzed a corpus of 1,489 projects to determine the severity thresholds for 9 test smells (Empty Test and Ignore Test are excluded from this calibration since they are binary by default). The calibration results are shown in Table 3.

As an example, Assertion Roulette has medium, high and very high severity set to 3, 5 and 10 respectively. These numbers represent the metric value (see Table 2): in the case of Assertion Roulette, it represents the number of assertions without descriptions. This means that if a method has a test smell value below 3, it should be considered not smelly (or in other words, it belongs to the best 70% of the corpus); if it has a test smell value between 5 and 10 it should be considered as high severity (it belongs to the worst 20% methods of the corpus); above a value of 10 it should be considered as very high severity, since it belongs to the worst 10% of the corpus.

As we can notice, 5 test smells have severity equals to 0: this means that the metric by which the smell is measured is so rare that even by considering the 90_{th} percentile we obtain a severity value of 0. Hence, for these cases, any test having a metric value higher than 1 results in a classification of very high severity. The distribution of the metrics across the observed projects for each test smell can be found in the replication package [28].

Table 3: Severity thresholds calculated from the benchmark

Test Smell	Severity Threshold		
	Medium	High	Very High
Assertion Roulette	3	5	10
Eager Test	4	7	39
Verbose Test	13	19	30
Conditional Test Logic	0	1	2
Magic Number Test	0	0	1
General Fixture	0	0	0
Mystery Guest	0	0	0
Resource Optimism	0	0	0
Sleepy Test	0	0	0

Previous studies have empirically investigated to what extent test smells are spread in software systems, by analyzing the distribution of test smells in source code [7, 10]. However, since our derived thresholds are higher, the diffusion of test smells would decrease.

As a first step to understand the test smells diffusion in open source projects using the new derived thresholds, we re-run the test smell analysis on our corpus of 1,489 projects using the new aforementioned thresholds. In Figure 3 we present the result. As we might expect since the new thresholds are stricter, we obtain from 8% to 30% less test smells instances than when using the old thresholds. To notice that the diffusion of the old thresholds are in line with what previous studies found [7]: "Assertion Roulette" is present in $\approx$ 50% of the test classes and "Eager Test" in 30% of them. However, by applying the new thresholds we obtain that "Assertion Roulette" is present in $\approx$ 30% of the classes and "Eager Test" in $\approx$ 15% of them. As for "Conditional Test Logic" and "Verbose Test" we could not find previous literature on their diffusion, however from the figure we can see that their occurrence is much lower when using the new thresholds.

In Table 4 we present the result for every test smell. Given that we defined new thresholds for only four smells, the numbers for the others are identical. Since out of the scope of this paper, we did not further investigate the test smells diffusion on OSS systems. New research should be carried on this topic, for example by investigating the co-occurrence of the smells, similar to what previous studies did with the old thresholds [7].

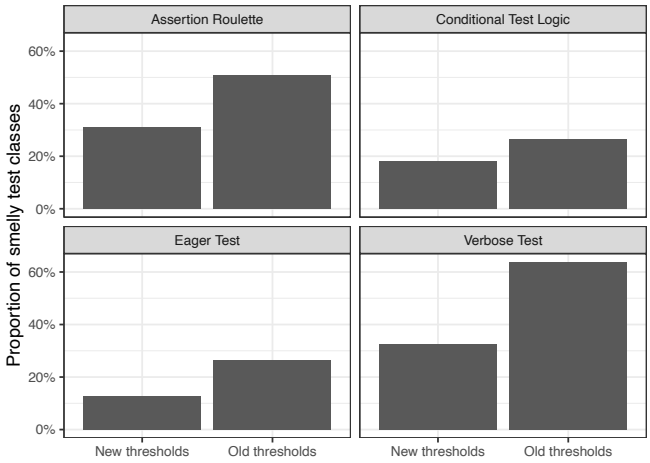


Figure 3: Diffusion of test smells across OSS systems: new thresholds *vs* old thresholds (N=1,489)

Thresholds' evaluation. After identifying the new thresholds, we want to determine if they are also aligned with the developer's perception of how much impact these instances have on the test suite maintainability. To this aim, we modify TSDETECT to incorporate the defined thresholds and plug it in BCH. Using a survey (explained in Section 3.5), we ask the developers their perception of the shown test smell severity by using a Likert Scale [16]. Then we triangulated our severity score to the users' perception of severity employing the Spearman's rank correlation coefficient. We use the Spearman's rank correlation test, as we could not make any

Table 4: Test smells' distribution across systems (N=1,489)

Test Smell	Old Thresholds		Derived Thresholds	
	Impacted	Not impacted	Impacted	Not impacted
Assertion Roulette	56,177 (50.9%)	54,278 (49.1%)	34,156 (30.9%)	76,299 (69.1%)
Cond. Test Logic	29,077 (26.3%)	81,378 (73.7%)	20,095 (18.2%)	90,360 (81.8%)
Empty Test	1,280 (1.2%)	109,175 (98.8%)	1,280 (1.2%)	109,175 (98.8%)
General Fixture	8,829 (8.0%)	101,626 (92.0%)	8,829 (8.0%)	101,626 (92.0%)
Mystery Guest	6,071 (5.5%)	104,384 (94.5%)	6,071 (5.5%)	104,384 (94.5%)
Sleepy Test	4,500 (4.1%)	105,955 (95.9%)	4,500 (4.1%)	105,955 (95.9%)
Eager Test	29,333 (26.6%)	81,122 (73.4%)	14,284 (12.9%)	96,171 (87.1%)
Ignored Test	3,105 (2.8%)	107,350 (97.2%)	3,105 (2.8%)	107,350 (97.2%)
Resource Optimism	7,345 (6.6%)	103,110 (93.4%)	7,345 (6.6%)	103,110 (93.4%)
Magic Num. Test	18,920 (17.1%)	91,535 (82.9%)	18,920 (17.1%)	91,535 (82.9%)
Verbose Test	70,461 (63.8%)	39,994 (36.2%)	36,080 (32.7%)	74,375 (67.3%)

assumptions about the distribution of our data, thus ruling out the use of Pearson's test [9].

In Table 5 we show the results: all the four test smells for which we defined non-binary severity thresholds have a statistically significant difference between our proposed severity and user-perceived maintainability impact. Furthermore, Verbose Test and Conditional Test Logic showed a high statistically significant relationship ($p < 0.001$) and a strong Spearman's coefficient ($0.6 \leq r_s \leq 0.79$), while Eager Test and Assertion Roulette showed a lower statistically significant relationship ($p < 0.05$) and a weaker Spearman's coefficient ($0.2 \leq r_s \leq 0.39$).

Table 5: Spearman's rank correlation between test smell severities as set by our thresholds and rated by users. Statistically significant results are in bold.

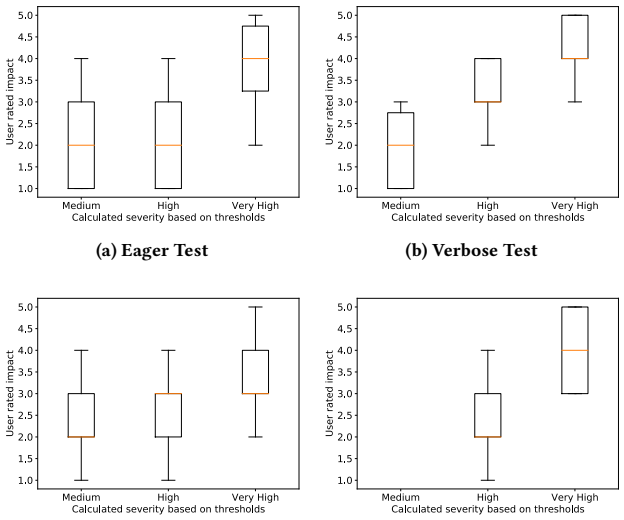
Test Smell	Responses	p	r_s
Eager Test	42	**0.027**	0.342
Conditional Test Logic	36	**<0.001**	0.753
Verbose Test	51	**<0.001**	0.679
Assertion Roulette	47	**0.016**	0.350

Figure 4 shows the distribution of the user-submitted impact ratings for each test smell severity category. As previously discussed, we can notice that for Verbose Test and Conditional Test Logic, the ratings submitted by the users are aligned with our thresholds, and the relationship is strong. For Eager Test and Assertion Roulette the difference is less visible instead, though statistically significant. Across the four smells, the threshold that aligns better with developers' perceptions is 'Very High', thus suggesting that it is the most appropriate to be used with practitioners.

4.2 RQ2: Developers' Perceptions

In this RQ, we want to investigate developers' overall perception on test smells found in *their* codebase: to this aim, we asked them to indicate for each test smell instance whether they would classify it as a valid instance to remove, how much priority they would give to the refactoring, and how long it would take according to them.

Refactoring evaluation. In Table 6 we show the percentage of actions the surveyed users indicated to take for each test smell instance identified in the BCH analysis report. We got a total of 301 responses. In Table 7 we show the impact ratings for test smell



(a) Eager Test (b) Verbose Test

(c) Assertion Roulette (d) Conditional Test Logic

Figure 4: Test smell severity vs. user rated impact values

Table 6: Actions taken by users, by identified test smell

Test Smell	Responses	Immediate Refactor	Long-term Refactor	No Action
Empty Test	14	71.43%	21.43%	7.14%
Sleepy Test	13	61.54%	23.08%	15.38%
Mystery Guest	10	60.00%	10.00%	30.00%
Resource Optimism	12	58.33%	16.67%	25.00%
Cond. Test Logic	39	53.85%	33.33%	12.82%
Verbose Test	54	48.15%	40.74%	11.11%
Magic Number Test	23	47.83%	34.78%	17.39%
Assertion Roulette	55	29.09%	50.91%	20.00%
Eager Test	51	21.57%	50.98%	27.45%
Ignored Test	10	20.00%	50.00%	30.00%
General Fixture	20	20.00%	40.00%	40.00%

Table 7: Impact ratings on test suite maintainability

Test Smell	Responses	Very Low	Low	Medium	High	Very High
Assertion Roulette	47	17.0%	27.7%	38.3%	14.9%	2.1%
Conditional Test Logic	36	2.8%	25.0%	30.6%	22.2%	19.4%
Eager Test	42	31.0%	26.2%	21.4%	16.7%	4.8%
Empty Test	13	0.0%	15.4%	7.7%	46.2%	30.8%
General Fixture	13	23.1%	30.8%	38.5%	7.7%	0.0%
Ignored Test	8	12.5%	12.5%	12.5%	37.5%	25.0%
Magic Number Test	20	15.0%	20.0%	30.0%	30.0%	5.0%
Mystery Guest	8	0.0%	12.5%	37.5%	50.0%	0.0%
Resource Optimism	10	10.0%	0.0%	20.0%	60.0%	10.0%
Sleepy Test	11	0.0%	9.1%	18.2%	63.6%	9.1%
Verbose Test	51	13.7%	23.5%	23.5%	27.5%	11.8%

instances which were rated as either short or long term refactoring candidates (no action excluded).

Smell evaluation. We now discuss the finding for each test smell separately, taking into account the optional additional remarks field (**SQ4**) of the survey, where the user could add remarks regarding their perception of the smell.

Assertion Roulette: Assertion Roulette is rated as having a low to medium impact on test suite severity and considered primarily as long term technical debt. One reason to mark assertion roulette as a no action item is that the test method name accurately reflects the reason for the test to fail and thus no further comments in the assertions are required. In some instances, the purpose of a block of assertions was described in a comment, which would require looking at the source code and line number to see why the test failed.

Conditional Test Logic Conditional Test Logic was primarily rated as a short-term refactoring candidate having medium to high impact. Of the 39 reported cases, 24 times it was reported as immediate refactoring, 13 as long-term refactoring and 2 times as no actions. In the comments, some developers stated that they use conditions to avoid running the test in a specific configuration (*e.g.*, on Windows, or using a specific version of the library, *etc.*), and in this case is difficult to avoid an if-else condition.

Eager Test: Eager Test is considered to have a low impact on test suite maintainability and is considered primarily as technical debt which can be addressed later or will not be addressed at all. Of the 51 reported cases, 26 times (50.98%) was reported as long-term refactoring, and 14 times was reported as no action/false positive. Only in 11 cases was considered as immediate refactoring.

Among the "no action" feedback, a repetitive comment is that getters and setters should be left out from the analysis. In this case, by getters and setters we not only mean the Java encapsulation pattern (*i.e.*, getters and setters of class members), but all the methods that retrieve (*e.g.*, get) or set a variable in the production class. Listing 1 exemplifies a common won't fix Eager Test pattern.

```
1   @Test
2   public void testFooAction() {
3       Foo foo = new Foo(...);
4       foo.setProp1(...); //1st production method call
5       foo.setProp2(...); //2nd call
6       int result = foo.action(); //3rd call
7       assertEquals(result, expectedResult);
8       assertEquals(foo.getProp1(), ...); //4th call
9       assertEquals(foo.getProp2(), ...); //5th call
10  }
```

Listing 1: Example of a test case containing an Eager Test

Empty Test Empty test is considered as an immediate refactoring candidate and rated as having a high to very high impact on test suite maintainability. In this case, almost all the developers agreed that the test should be immediately removed, since it does not contain any assert statement.

General Fixture General Fixture is considered among the sampled developers as having a low to medium impact on the test suite maintainability and not considered as an issue to remove. In most of the cases, the tests used all or a different combination of the test fixture variables – on class level, all of the test fixtures were used.

Ignored Test Ignored Tests are considered to have a high impact on test suite maintainability and considered for long-term removal. Ignored tests are often accompanied by a bug ID inside the ignore statement (*e.g.*, `@Ignore("PROJECTCODE-123")`). Developers consider that these tests should not count as a test smell instance if they are properly documented in the bug tracking system. The source of the bug might not be in the test but rather in the production code.

Magic Number Test Magic Number Test is considered low to medium severity and rated primarily as a short-term refactoring candidate. Tests exhibiting the Magic Number Test which were marked for refactoring were accompanied by comments from the developers, such as where the numbers come from and how they could be replaced with named constants (e.g., HTTP status codes). In instances marked as will not fix, the magic numbers were the results of a calculation performed inside the tested function. In these cases, the developers do not see the problem of not naming the constants, as the value according to them should be clear from the test context.

Mystery Guest Mystery Guest is rated as having medium to high impact on test suite maintainability and is considered to be primarily a candidate for immediate refactoring. Unless the developer wants to test the reading/writing of a file to the file system, a more appropriate approach would be to create a data structure that holds the necessary information.

Resource Optimism Resource Optimism is considered to be primarily a short-term refactoring candidate of high severity. The refactoring for this test smell can be done by adding file existence checks for each resource used.

Sleepy Test Sleepy Test is primarily perceived as having a high impact on test suite maintainability and is considered for immediate refactoring. The presence of thread sleep calls indicates that the test is not a good unit test and negatively impacts the test suite maintainability, since it might introduce flakiness. The removal of this test smell could require rewriting the entire test.

Verbose Test Verbose Test is split between immediate and long term refactoring. It is rated as having an overall medium to high impact on test suite maintainability. In the feedback, the developers pointed out that the tests exhibiting Verbose Test smell can be either split into multiple smaller tests or refactored to use methods to decrease the test size.

Analysis of '`will not fix`' reports To gain more insights on why developers reported test smells as '`will not fix`', we examine all of the instances which had their source code accessible in a public repository (29 out of 60 ratings). We performed an open card sorting method [20] with two inspectors. This method allows creating mental models and allowing the definition of taxonomies from input data [20].

The two inspectors were the first two authors of this paper, each with over four years of Java development experience. We applied the open card sorting method by first independently grouping the test smell instances and then comparing the results with a discussion about the classification and group differences. After we have agreed on a unified view on the classification, we move to the next test smell. The inspectors had at their disposal the source code of the test and the comments provided by the developer.

Based on the open card sorting methodology, we have identified the following four categories of will not fix test smells.

False Positive. These instances are true false positives – they show the limitations of our current tooling and approach. Here the fix would require modifying TSDETECT or creating a new tool.

The issue could not be fixed by setting better thresholds. There were a total of 7 false positives (24%).

Dismissed, Hard to fix. In this category, developers either do not see the problem with the test or changing the test to remove the smell instance would require a significant amount of effort to fix. The fix in some instances would require rewriting the production class under test, or even the related classes to support better testability. In these cases, the test smell points to larger problems of the codebase, not just related to the test design itself. There were a total of 11 instances (38%).

Dismissed, Easy Fix. Test smells in this category can be easily removed by refactoring them. However, developers either do not acknowledge them as a problem or are unwilling to refactor it to remove the test smell instance. In all identified cases in this category, the readability and maintainability of the test would be significantly improved if the test was refactored to follow the best practices. From the developer's point of view, these might be only perceived to be marginal gains and they concentrate their efforts elsewhere. There were 9 instances (31%) in this category.

Acknowledged problem, won't fix. Test smell instances in this category are acknowledged by the developers as possibly problematic, but in the domain context, they are perceived to be an acceptable trade-off between maintainability and ease of writing. The two instances of this category were Assertion Roulette smells. Here the developers consider "simple" assertions to be self-documenting. For example, in case of failing '`assertEquals( HttpStatus.BAD_REQUEST, ...);`' is easy to understand, even without documentation.

5 DISCUSSION AND FUTURE WORK

In this section we discuss our results, their implications, and future research directions.

Severity Rating of Test Smells. Our approach of defining test smell severity shows promising results when applied to certain test smells, as pointed out by the developers agreeing with our rating. On the other hand, for some test smells our approach does not work, as they are very rare, and an alternative classification would have to be used to define their severity.

We defined new severity thresholds for Assertion Roulette, Conditional Test Logic, Eager Test, and Verbose Test. These test smells had their respective metrics distributed across the selected benchmark project's codebase. On the contrary, several of our proposed thresholds had zero-valued thresholds up to or including the very high severity, which presents problems as we can not make any distinction between the severity levels. A possible approach to introduce granularity for the test smells where the thresholds are zeros would be to investigate the top 10 percentile of the worst projects, and extend the scale to there. Then set the high and very high thresholds based on the values of applying the original methodology on only this filtered sample size again. This would ensure that the medium risk threshold is at zero following the original methodology, and would introduce more fine-grained values for the high and very high thresholds.

Since out of the scope of this paper, we treated the smells with zero thresholds as critical (very high) severity, because present in less than 10% of the benchmark corpus. However, in certain test smells such as Sleepy Test, the usage of even one thread sleep call versus multiple calls can be considered to be irrelevant, as even one such call makes the test dependent on the timing and can introduce multi-threading faults and flakiness.

Two test smells were excluded from the calibration: "Ignored Test" and "Empty Test". For these test smells, no conventional metric exists; hence an alternative should be proposed. For example, "Ignored Test" smell can be classified based on whether the @Ignore annotation contains a bug tracking issue identifier or description for the failing test. Developers can then focus on tests that are ignored without a description to investigate the root cause of the failure.

Classification of Empty Test could involve looking at whether the test is empty and does not contain any commented code. If it contains commented code, this could indicate that the test should have an `@Ignore` annotation instead, since it is masking an uncovered issue. In the latter case, these tests could be considered to be of high severity and the developers should investigate why the test code was commented out.

Developers' perception. By using the new derived thresholds we lowered the number of false positives the tool detects. However, as we saw from our RQ2, developers in some cases still do not perceive the smell as an actual problem and mark it as no-fix. The reason for this could be that the thresholds are still set too low, and most of the found instances are perceived as not problematic. This result is in line with previous studies [30], where the authors showed that often developers do not perceive test smells as actual problems. More recently, De Bleser *et al.* [10] studied developers' perception on test smells of Scala projects. The authors found that General Fixture and Mystery Guest are the test smells that are perceived as the most problematic. While we can confirm this result for Mystery Guest, the developers that participated in our study rated General Fixture as having a low impact on test suite maintainability.

However, when analyzing the cases in which the developers rated an instance as a no-fix (Section 4.2), we saw that in few cases the tool had false positive (*e.g.*, getters and setters should not be counted as method calls), but in the majority of the cases they dismissed the test smell instance as they were unwilling to refactor the test (even if it was an easy fix). Even though by fixing the smell the readability and maintainability would be significantly improved, these might be only perceived to be marginal gains, and they concentrate their efforts elsewhere. In this case, increasing the thresholds would not change their perception.

Test Smell Triage. Test smells density, defined as the number of test smells in a test class, is too low level to be used as direct metrics to present to developers. It can be incorporated into another higher level metric to measure test code quality.

For example, a possible approach would be to look into test smell co-occurrence, combined with our proposed severity levels, to rank individual unit tests, test files, or test modules as ones that need more attention than the others. A test file impacted by several different instances of high severity test smells might need more attention than a file impacted by only one test smell type of low severity. This would give better actionability to the developers, as they would then have an indicator where to direct their attention. This approach would require investigating whether a combination

of several test smells of different severity levels is worse than, for example, one very high severity test smell.

New or Improved Test Smell Detection Tool. We have identified several limitations in how tsDetect identifies test smells and that the current definition of certain test smells does not match the developer's perception or common practices. This mismatch between the developer's perception of what is vs. is not a test smell cannot be fixed by setting thresholds alone. In the case of Assertion Roulette, the developers consider good test naming to be sufficient to document the assertions. Incorporating or creating a test naming model, such as the one proposed by Meester [18] for rating how well a method is named, could be used to determine if a test is named correctly and thus the assertions do not need an additional explanation. Currently, no tool also considers block comments above a series of assertions as a description, while developers consider this to be enough to describe what the assertions are for.

For Ignored Test, further research could be investigating whether it is possible to classify the description in the `@Ignore` annotation to determine if it contains a bug tracking issue identifier or a description for why it is failing. The impact of Eager Test detection excluding getters and setters could be investigated to see if this detection method would improve the probability that flagged Eager Tests are given bigger priority by the developers. All of the above could be further explored and then integrated into a new test smell detection tool, which would detect a smaller subset of test smells; however, the smells detected would be more likely to be perceived by the developers as problematic and an issue to remove.

6 THREATS TO VALIDITY

Construct validity. Threats to construct validity concern our research instruments. We used the tool tsDetect to detect test smells and classify them according to our defined thresholds. For the subset of test smells we observed, the tool was reported to have an F-score surpassing 87% for each test smell as determined by a comprehensive review of multiple projects [23].

Internal validity. Threats to internal validity concern factors that could affect the variables and the relations being investigated. All of the contacted developers for this experiment were users of BCH, who have used it at least once in the past. The developers using a code quality tool might have more knowledge about code quality than developers who have never used such tools and thus their opinion might be different than those of developers who do not use BCH or any other quality control tool. Furthermore, developers that use code quality tools might care more of code quality (hence rating a test smell as high impact) than developers that do not use code quality tools.

External validity. Threats to external validity concern the generalization of results. We are aware that derivation of metrics thresholds from a benchmark dataset introduces dependency on the dataset and its representativeness. Further investigation needs to be done to determine the degree of sensitivity of this approach with respect to different benchmark datasets, precomputed metrics, and different metric extraction tools. However, to mitigate this issue, we used a big corpus of 1,489 OSS systems of different scopes, size and characteristics, to strengthen the generalizability of our findings.

For the second part of the study, we modified BCH by including a test smell detector and asked developers to rate their code. We have received responses from 31 developers, who evaluated 47 distinct projects. In comparison to other experiments in software engineering, our sample size is above the median (30 respondents) [26].

We only analyzed Java source code for test smells. The thresholds we defined would have to be defined per language, as the occurrence of test smells might be different across various languages and unit testing frameworks. The methodology for the test smell calibration should nonetheless generalize to other languages if they are supported by a test smell detection tool, which can also provide metrics in addition to detection.

7 CONCLUSION

In this paper, we investigated the classification of test smells based on their severity. To do this, we define metrics for each test smell and then apply the benchmark-based threshold derivation on a sample of open-source projects. The result of this process allows us to give test smells a severity rating, from low to very high, which we verified with a sample of developers using their test source code to see if they agree with the rating classification. For four test smells (Assertion Roulette, Eager Test, Verbose Test and Conditional Test Logic), we defined non-binary severity thresholds and have a statistically significant difference between our proposed severity and user-perceived maintainability impact. Furthermore, Verbose Test and Conditional Test Logic showed a high statistically significant relationship and a strong Spearman coefficient.

We conducted a study of developer perception on selected test smells in their codebase using a modified version of BCH that allows for the detection of test smells with the new derived thresholds. We asked the developers to indicate for each test smell instance whether they would classify it as a valid instance to remove, how much priority they would give to the refactoring, and how long it would take according to them. Among the main results, we saw that Empty Test and Sleepy Test are considered the smells with the highest priority in refactoring, follows by Mystery Guest and Resource Optimism. The smells with the lowest priorities instead were General Fixture, Ignored Test, and Eager Test.

Furthermore, we analyzed test smells the developers marked as won't fix, trying to discover the reasons behind it, indicating how test smell detection tools could be improved to match the developer's views on what is a test smell. This information can also be used to enhance the proposed test smell severity ratings.

Furthermore, we have shown that test smell detection can be successfully integrated into a code quality tool to inform developers on test issues and help make tests more maintainable.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 642954. Alberto Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] Efthimia Aivaloglou and Felienne Hermans. 2016. How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 53–61.

[2] Tiago L. Alves, Christiaan Ypma, and Joost Visser. 2010. Deriving Metric Thresholds from Benchmark Data. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609747

[3] Mauricio Aniche, Christoph Treude, Andy Zaidman, Arie Van Deursen, and Marco Aurelio Gerosa. 2016. SATT: Tailoring Code Metric Thresholds for Different Software Architectures. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 41–50. https://doi.org/10.1109/SCAM.2016.19

[4] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering* 40, 11 (11 2014), 1100–1125. https://doi.org/10.1109/TSE.2014.2342227

[5] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. 2012. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal* 20, 2 (2012), 287–307. https://doi.org/10.1007/s11219-011-9144-9

[6] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (8 2015), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0

[7] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 56–65. https://doi.org/10.1109/ICSM.2012.6405253

[8] Manuel Breugelmans and Bart van Rompaey. 2008. TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites. In *1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT-1)*.

[9] Wayne W. Daniel. 1990. *Applied nonparametric statistics*. PWS-Kent.

[10] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Assessing diffusion and perception of test smells in scala projects. *IEEE International Working Conference on Mining Software Repositories* 2019-May (2019), 457–467. https://doi.org/10.1109/MSR.2019.00072

[11] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. 2015. Automatic metric thresholds derivation for code smell detection. *International Workshop on Emerging Trends in Software Metrics, WETSoM* 2015-August (2015), 44–53. https://doi.org/10.1109/WETSoM.2015.14

[12] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[13] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 1–11. https://doi.org/10.1109/ICST.2018.00011

[14] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2013. Automated Detection of Test Fixture Strategies and Smells. In *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*. 322–331. https://doi.org/10.1109/ICST.2013.45

[15] Maudlin Kummer. 2015. *Categorising Test Smells*. Bachelor Thesis. University of Bern.

[16] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[17] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[18] Sander Meester. 2019. *Towards better maintainable software: creating a naming quality model*. Master Thesis. University of Amsterdam.

[19] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[20] Hazel E. Nelson. 1976. A modified card sorting test sensitive to frontal lobe defects. *Cortex* 12, 4 (December 1976), 313–324.

[21] Fabio Palomba, Gabriele Bavota, Massimilliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 101–110. https://doi.org/10.1109/ICSME.2014.32

[22] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic Test Smell Detection Using Information Retrieval Techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 311–322. https://doi.org/10.1109/ICSME.2018.00040

[23] Anthony Peruma. 2018. *What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications*. Master Thesis.

[24] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19)*. IBM Corp., USA, 193–202.

[25] Abdus Satter, Nadia Nahar, and Kazi Sakib. 2017. Automatically Identifying Dead Fields in Test Code by Resolving Method Call and Field Dependency. In *5th International Workshop on Quantitative Approaches to Software Quality*. 51–58.

[26] Dag Sjøberg, Jo Hannay, Ove Hansen, V.B. Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and A.C. Rekdal. 2005. A survey of controlled experiments in software engineering. *Transactions on Software Engineering* 31 (10 2005), 733–753. https://doi.org/10.1109/TSE.2005.97

[27] Dag I. K. Sjøberg, Aiko Fallas Yamashita, Bente Anda, Audris Mockus, and Tore Dybå. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *Transactions on Software Engineering* 39 (12 2013), 1144–1156. https://doi.org/10.1109/TSE.2012.89

[28] Davide Spadini. 2020. Replication package. https://doi.org/10.5281/zenodo.3611111

[29] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–12. https://doi.org/10.1109/ICSME.2018.00010

[30] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, 4–15. https://doi.org/10.1145/2970276.2970340

[31] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2001)*. 92–95.

[32] Bart van Rompaey, Bart du Bois, and Serge Demeyer. 2006. Characterizing the Relative Significance of a Test Smell. In *22nd IEEE International Conference on Software Maintenance*. IEEE, 391–400. https://doi.org/10.1109/ICSM.2006.18

[33] Bart van Rompaey, Bois du Bois, Serge Demeyer, and Matthias Rieger. 2007. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *Transactions on Software Engineering* 33, 12 (December 2007), 800–817. https://doi.org/10.1109/TSE.2007.70745

[34] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, 385–396. https://doi.org/10.1145/2610384.2610404