2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)

Do the Test Smells *Assertion Roulette* and *Eager Test* Impact Students' Troubleshooting and Debugging Capabilities?

Wajdi Aljedaani*, Mohamed Wiem Mkaouer†, Anthony Peruma‡ and Stephanie Ludi*
*University of North Texas. Email{wajdi.aljedaani, Stephanie.Ludi@unt.edu}
†Rochester Institute of Technology. Email{mwmvse@rit.edu}
‡University of Hawaii at Manoa. Email{peruma@hawaii.edu}

***Abstract*—To ensure the quality of a software system, developers perform an activity known as unit testing, where they write code (known as test cases) that verifies the individual software units that make up the system. Like production code, test cases are subject to bad programming practices, known as test smells, that hurt maintenance activities. An essential part of most maintenance activities is program comprehension which involves developers reading the code to understand its behavior to fix issues or update features. In this study, we conduct a controlled experiment with 96 undergraduate computer science students to investigate the impact of two common types of test smells, namely *Assertion Roulette* and *Eager Test*, on a student's ability to debug and troubleshoot test case failures. Our findings show that students take longer to correct errors in production code when smells are present in their associated test cases, especially *Assertion Roulette*. We envision our findings supporting academia in better equipping students with the knowledge and resources in writing and maintaining high-quality test cases. Our experimental materials are available online[1]**

***Index Terms*—Test smells, unit testing, software engineering education, computer science education, software testing**

I. Introduction

An essential activity in ensuring the quality of a software system is unit testing, where developers write code to verify the behavior of the implemented system's production (i.e., source) code [35]. By using unit tests, organizations and project teams automate the discovery of flaws in their system that would otherwise go unnoticed or consume developer time [25]. Given this invaluable benefit in improving the overall quality of a software system, many projects and organizations mandate that developers write unit tests as part of their software development process [29], including the adoption of a test-driven development approach [12].

However, as with production code, test code is also subject to bad programming practices by developers, known as test smells [41]. Likewise, similar to code smells, test smells are also an indicator of deeper problems, such as bad design or implementation choices in the test suite. Prior research has shown that test smells negatively impact the system's maintainability. Specifically, test smells have been shown to increase the change- and defect-proneness of the system's codebase [38], increase the flakiness of test cases [14], and negatively impact test code readability and understandability [41]. Furthermore, developers' injection of these test smells ranges from lack of proper testing discipline (i.e., mistakes/-carelessness and non-removal of debugging code) to testing knowledge gaps [33].

As described above, through a series of empirical studies and developer interviews, the research community has shown that test smells impact a system's internal quality, thereby impacting maintenance activities. To further validate these findings and expand the body of knowledge on test smells, our study investigates the effect of test smells on code comprehension activities. To this extent, we conducted a sizeable human-based study with undergraduate students enrolled in a computer science program at a university in North America.

Similar to code smells, there are multiple types of test smells defined in published literature [3]. However, as this is a human-based study and involves students, examining each smell in the test smells catalog is not feasible due to time constraints. Therefore, in this study, we focus our analysis on only two smell types– *Assertion Roulette* and *Eager Test*. We arrived at these two smell types by reviewing multiple studies [10], [24], [33], [39] and comparing the distribution of smell types that these studies have in common; both of these smell types frequently occur in the test suites of open-source Java systems.

A. Motivation & Goal

While there exist studies that evaluated the impact of test smells on the code comprehension capabilities of students, these studies either involved students writing complete test cases [8], [9], [13] or evaluating the test suites from large, well-established open-source systems with which they have no prior experience [11]. In contrast, as we elaborate in Section III, our work involves students examining pre-written test cases corresponding to simplistic use cases and making updates to the production (i.e., system under test) code to correct failing test cases. Hence, to a large extent, we eliminate any influence placed on the student's cognitive load caused by understanding (or being overwhelmed by) the overall behavior and technical design/architecture of a complex and unfamiliar system. Therefore, the findings from our study are more closely aligned with the actual impact of test smells on code comprehension. Furthermore, this study also allows us to compare our findings against the work by Bai et al. [7], which states that *Assertion Roulette* should not be considered a bad smell for students.

In this study, our goal is to determine *the extent to which*

[1]https://wajdialjedaani.github.io/testsmellstd/

2832-7578/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-SEET58685.2023.00009

the presence of test smells in the test suite impacts a student's troubleshooting and debugging capabilities. Specifically, our work compares the effect that smelly and non-smelly test suites have on students when tasked with fixing failing test cases by only correcting defects in the production code of a system they are familiar with. We theorize that test files exhibiting test smells cause an increase in code comprehension time than those without test smells, resulting in students spending more time on maintenance activities.

B. Contribution

The results of our study show that test smells negatively impact a student's code comprehension capabilities. Furthermore, the *Assertion Roulette* smell causes students to take more time to address issues in the production code than the *Eager Test* smell. Our study highlights the need for academia to invest in and prioritize teaching students about all types of test smells, their harmful impact on maintenance activities, and tools that can be used to automatically detect and eliminate these smells from the test suite of a software system.

II. Test Smell Definitions & Related Work

This section provides definitions of the two test smell types (i.e., *Assertion Roulette* and *Eager Test*) utilized in our study and an overview of the related work in this area.

A. Test Smell Definitions

Assertion Roulette. The passing/failing of a test case is determined by the execution of the assertion method it contains. These assertion methods allow developers to include an optional textual message indicating the reason for the failure of the assertion. This smell occurs when a test method contains two or more assertions without an explanation message. Troubleshooting the failure of a test case becomes challenging as the developer is unaware of the cause of the failure.
Eager Test. This smell occurs when a test method verifies multiple functionalities of the production code by invoking several production methods. This smell makes it hard to understand the true purpose of the test. Furthermore, it increases the coupling between the test method and production code, which, in turn, negatively impacts maintenance.

B. Related Work

Several automated techniques and tools for detecting test smells have been published in the literature [3]. In addition, researchers have identified a collection of test smells [21], while others have concentrated on the effects of test smells and removal techniques [30]. For example, Van Bladel and Demeyer suggested eliminating test smells in the context of refactoring test code [40], and Van Deursen et al. highlighted harmful test smells and techniques to eliminate them [41]. In addition, they offered conceptual and technical explanations for evaluating students' activity by identifying test smells in their code and offering test smell-related observations. This section highlights several prior studies that particularly shaped our methodology. Next, we divide the related work into two different aspects of test smell in education: software testing in education, where we focus on current approaches used to examine the testing in education; Students' programming and testing activities, which focus particularly on unit testing inside the classroom. Finally, Table I presents a summary of the systematic analysis studies in the related work.

1) Software Testing in Education

Educators have examined a variety of assessment integration strategies for computer science courses. For instance, some researchers instruct students to submit their software tests and solutions [19], [23], while others involve students in peer testing [36], [22]. Fraser et al. [20] proposed a Code Defenders game that is utilized to engage students' activities in the test suite. Aniche and colleagues [4] conducted a survey involving 84 first-year computer science students regarding the difficulties associated with learning software testing. They also investigated the errors that were made in the lab work of 230 students. According to their findings, there are eight different types of typical errors. These include test coverage, maintainability of test code, understanding of testing principles, boundary testing, state-based testing, assertions, mock objects, and tools.

Previous studies have investigated both the level of quality of student-written test code [4], [15], [16], [18] and the viewpoints of students on unit testing [4], [22]. These studies utilize several metrics, such as the frequency of defects identified by student-created test cases and branch coverage. To evaluate the incremental testing procedures of software development projects, Kazerouni et al. [28] developed new metrics: the balance and sequencing of the testing effort. According to an evaluation conducted by Carver and Kraft [15], students in the senior year of computer science do not have the skills necessary to make good use of test-coverage techniques. These experiments were performed in a classroom setting, and participants were given grades for their participation.

2) Students' programming and testing activities

Several approaches [2], [17] have been developed for assessing student-created test suites. Bai et al. [8] studied the impact of a checklist on the students writing test cases. The students anticipated that they would develop JUnit tests to check the functionality of a program that had been implemented to evaluate the student completeness, effectiveness, and maintainability. In recent work, Buffardi and Aguirre-Ayal [13] analyzed students' work on testing assignments to examine their adoption of test smells. The authors also investigated the relationship between three types of test smells and the test accuracy of the students' work. Bai et al. [9] performed an experimental study to learn how students understand unit testing and what obstacles they face when engaging in unit testing. Bavota et al. [11] performed a control experiment on students and industrial developers on six test smells. They execute software comprehension tasks on test suites with and without test smells and measure performance using correctness and time. Participants cannot conduct the necessary maintenance when this test smell is present. Thus, the smell significantly

TABLE I: Summary of the systematic analysis studies in related work.

Study	Year	Purpose	Evaluation	Test Smell	Participant	# of Participants
[11]	2015	Understanding how test smells is spread in real software systems	Experiment	TCD, MG, GF, ET, LT, AR, IT, SE	Students Developers	49 12
[20]	2019	Engaging students with software testing in an entertaining way	Survey	Game experiment	Students	123
[13]	2021	Exploring smells exhibited by students first learning how to unit test	Unit tests, source code	MA, CL	Students	246
[9]	2021	Discovering students' perceptions & challenges when practicing unit testing	Survey	N/A	Students	54
[8]	2022	Assessing the impact of the testing checklist	Survey	N/A	Students	32

Abbreviation of test smells types: (AR) Assertion Roulette, (MG) Mystery Guest, (GF) General Fixture, (ET) Eager Test, (LT) Lazy Test, (IT) Indirect Testing, (SE) Sensitive Equality, (TCD) Test Code Duplication, (MA) Multiple Assertions, (CL) Conditional Logic.

affects it (bachelor students scored 48% correctness, while industrial developers scored 42%).

Researchers and educators commonly use test case/suite success rates to evaluate the quality of student-written source code [6], [27], [42]. In contrast, students' productivity is generally measured in lines of code per hour [5] or work session [27]. Unfortunately, there has been a lack of focus on the importance of test smells in the classroom. In a study similar to ours, Bai et al. [7] examined how the *Assertion Roulette* smell affects students' productivity and conduct while writing code. The authors employed the Bowling Score Keeper project, and the student was tasked with writing a Java app to calculate the score of a single bowling game according to a set of specifications and JUnit tests.

III. Study Design

The primary objective of this study is to assess to what extent *Assertion Roulette* and *Eager Test* hinder students' program comprehension. To do so, we evaluate the impact of these smells' existence on the debugging process when students use smelly test files to locate errors. Therefore, our main Research Question is:

> **RQ: To what extent do *Assertion Roulette* and *Eager Test* impact the time spent by students in debugging failing test cases?**

By *debugging time*, we mean the time needed for a student to troubleshoot a failing test method and fix its corresponding error in the production method under test. Based on existing studies, smelly test files hinder program comprehension, i.e., we hypothesize that students should take a longer time to locate an error raised by a smelly test method, as it is harder to read, in comparison to the time needed to find an error raised by a non-smelly test method. To address this hypothesis, we create a controlled experiment where we select one project, which already contains a unit test suite with 100% path coverage. Then, we inject errors in production methods, which are going to be caught by the tests, i.e., for each error injected into the production code, its corresponding test method is going to fail. For a set of created errors, the time needed to debug their failing test methods is expected to take longer if these test methods are smelly. This can be empirically validated if, for the same set of errors, the time needed to debug them would take longer if the test suite is smelly, in comparison with the debugging time when the same test suite is not smelly. Since we are interested in comparing the *debugging time*, out of a non-smelly test suite (referred to as *Suite N*), we create two variants of the same test suite: The first variant (referred to as Suite A) has the same testing logic of Suite N, but with test methods infected with the *assertion roulette* smell. Similarly, the second variant (referred to as *Suite E*) has *Suite N*'s methods infected with the *eager test*. These suites are described as follows:

- **Suite N**. It encompasses *(N)on-Smelly* test cases. Each production method is associated with one or multiple test methods, testing multiple scenarios and ensuring the coverage of all the method's execution paths. We followed the guidelines by XUnit [31].
- **Suite A**. We introduced *(A)ssertion Roulette* smell into *Suite N* test cases by testing multiple scenarios, for one given production method, under the same test method. This induces a test method with multiple assert statements, covering all the production method's execution paths. According to the literature, it hinders comprehension by making it difficult to determine which *assertion* has triggered the test failure [32].
- **Suite E**. We introduced *(E)ager Test* smell into *Suite N* test cases by testing multiple scenarios, for multiple production methods, under the same test method. This induces a test method with multiple assert statements, covering all the production method's execution paths. According to the literature, it hinders comprehension by making it difficult to determine which *method* has triggered the test failure [32].

To ensure each suite contains (or not) the intended test smell type, we run TS-Detect, one of the popular state-of-the-art test smell detectors [34], for each suite, as a sanity check.

A. Project Overview

Since we are measuring students' debugging time, we need to avoid any bias that can be introduced due to misunderstanding the program's behavior. Therefore, the chosen application should be intuitive for anyone to understand. For this purpose, we created a basic calculator application using Java programming language. The language was chosen to match students' familiarity with object-oriented programming at their level. Similarly, the choice of calculator is driven by intuitiveness and students' familiarity with its features under test. The calculator was designed with eight functions listed below:

- **Summation (Sum):** is a production method that takes as input two variables of type *double*, and returns their summation, e.g., `10 + 2 = 12`.

TABLE II: Summary of test cases for each category.

Method	# of Test Cases		
	Suite N Non-Smelly test	Suite A Assertion Roulette	Suite E Eager Test
Summation (Sum)	5	1	9
Subtraction (Sub)	5	1	
Multiplication (Mult)	7	1	
Division (Div)	5	1	
Square Root (SQRT)	5	1	
Modulo (Mod)	5	1	
Average (Avg)	4	1	
Factorial (Fact)	4	1	
Total	**40**	**8**	**9**

- **Subtraction (Sub):** is a production method that takes as input two variables of type *double*, and returns their subtraction, e.g., $10 - 2 = 8$.
- **Multiplication (Mult):** is a production method that takes as input two variables of type *double*, and returns their multiplication, e.g., $10 \times 2 = 20$.
- **Division (Div):** is a production method that takes as input two variables of type *double*, and returns their division, e.g., $10 \div 5 = 2$.
- **Square Root (SQRT):** is a factor that, when multiplied by itself, equals the original value of the given integer. A square root is represented by a radical symbol ($\sqrt{\ }$) and can be determined by the value of the power $\frac{1}{2}$ of an integer, e.g., $\sqrt{25} = 5$.
- **Modulo (Mod):** is the signed residue of a division, which occurs after dividing two numbers. It is computed by subtracting the divisor from the dividend until the resulting is less than the divisor, e.g., $5 \pmod 2 = 1$.
- **Average (Avg):** is a mathematical operation to compute the mean of a given set of numbers. It is the ratio of sum of all numbers in a given set to the number of values present in the set, e.g., avg of numbers present in set $A = \{1, 2, 3, 4, 5\}$ can be computed as $\frac{1+2+3+4+5}{5} = 3$.
- **Factorial (Fact):** is a function that outputs the product of all positive integers less than or equal to a given positive integer. It is indicated by an exclamation mark preceding that integer, e.g., $4! = 24$.

B. Test Suites Creation

For each production method, we need to create its corresponding test methods, ensuring 100% path coverage. These test methods were automatically generated by EvoSuite[2] and labeled as *Suite N*. Since the generated test methods' names are not descriptive, for each test method, we added a comment to indicate which production method it tests. It is critical for our experiments to ensure that the mapping between test and production methods is maintained. Otherwise, the overhead of students searching for such mappings would inflate the debugging time. To create *Suite A* and *Suite E*, we duplicate *Suite N*, and we manually introduce the smells based on their definitions that we outlined above.

To illustrate how these suites differ, Listing 2 shows 4 test methods from *Suite N*. Listing 3 shows how `test00()` and `test01()` (resp. `test05` and `test06`) are merged, since they are testing the same production summation (resp. subtraction) method. The merged methods have the *assertion roulette* smell, so they belong to *Suite A*. As for Listing 4, all methods from Listing 2 are merged into one test method, constituting the *eager test*. The merged method has the *eager test* smell, so it belongs to *Suite E*. This process has resulted in 40 test methods in *Suite N*, 8 test methods in *Suite A*, and 9 test methods in *Suite E*. The count of test methods for each test suite is summarized in Table II

C. Errors Generation

To create the errors, we used PITest[3], a Java mutation testing framework, to generate faults (or mutations) that are purposely seeded into the production methods. For a given mutated production method, if one or many of its corresponding test method(s) fail(s), then the error is caught. Otherwise, if the tests pass, then the error is missed. PIT is typically used to evaluate the quality of tests by the percentage of caught errors. In our context, we use PIT to generate arbitrary errors throughout the production methods. Then we selected errors while making sure each production method would have at least one error, and the selected errors were all caught by the 3 test suites.

Figure 1 presents two production methods, i.e., summation and subtraction, containing two errors. In the $sum\_$ method, the summation operation (+) is replaced with the subtraction operation (-), resulting in a faulty behavior. Likewise, the subtract operation (-) is replaced with the summation operation (+), in the $diff\_$.

```
public class Calculator {
    public double sum(double [] arr){
        //Creation of Array
        double sum_ = 0;
        for(int i =0; i < arr.length; i++){
            sum_ -= arr[i];
        }
        //adding all elements in an array
        System.out.println("Addition: "+sum_);
        return sum_;
    }

    public double subtract(double [] arr){
        //Creation of Array
        double diff_ = 0;
        for (int i = 0; i < arr.length;  i++){
            diff_ += arr[i];
        }
        //Subtracting all elements in an array
        System.out.println("Subtraction: "+
            diff_);
        return  diff_;
    }
}
```

Listing 1: Example for two production methods with seeded errors.

[2]https://www.evosuite.org/

[3]https://pitest.org/

D. Target Course

This experiment has been conducted in an undergraduate senior-level software engineering course[4]. Before joining this course, students have about two years of programming experience. This provides them with the background to perform the debugging needed in this experiment. Also, they are familiar with the process of searching for the root errors in a faulty code.

```
public class Calculator_ESTest extends
    Calculator_ESTest_scaffolding {
    //Test Cases for Sum Method
    @Test(timeout = 4000)
    public void test00() throws Throwable {
        Calculator calculator0 = new
            Calculator();
        double[] doubleArray0 = new double[2];
        double double0 = calculator0.sum(
            doubleArray0);
        assertEquals(0.0, double0, 0.01);
    }
    @Test(timeout = 4000)
    public void test01() throws Throwable {
        Calculator calculator0 = new
            Calculator();
        double[] doubleArray0 = new double[2];
        doubleArray0[0] = (-1.0);
        double double0 = calculator0.sum(
            doubleArray0);
        assertEquals((-1.0), double0, 0.01);
    }
    //Test Cases for Subtract Method
    @Test(timeout = 4000)
    public void test05() throws Throwable {
        Calculator calculator0 = new
            Calculator();
        double[] doubleArray0 = new double[2];
        double double0 = calculator0.subtract(
            doubleArray0);
        assertEquals(0.0, double0, 0.01);
    }
    @Test(timeout = 4000)
    public void test06() throws Throwable {
        Calculator calculator0 = new
            Calculator();
        double[] doubleArray0 = new double[5];
        doubleArray0[2] = 93.0;
        double double0 = calculator0.subtract(
            doubleArray0);
        assertEquals((-93.0), double0, 0.01);
    }
}
```

Listing 2: Example of test case source code for Non-test Smell test suite.

E. Pilot Study

A pilot study is the initial phase of the entire research protocol and is generally a smaller-scale study that serves to solidify the main study [26]. Therefore, prior to the primary investigation, we conducted a pilot study with four undergraduate students who were later excluded from the controlled experiment. The goal of the pilot study was to guarantee that the experiment's instructions were clear and to establish an approximate duration for the lab session. Following the pilot study, we decided to provide all upcoming participants with documentation on how to set up the programming environment. Also, we would only allow students to participate in the experiment when they have their environment ready to avoid skewing our measurements.

```
public class Calculator_ESTest extends
    Calculator_ESTest_scaffolding {
    //Test Cases for Sum Method
    @Test(timeout = 4000)
    public void test00() throws Throwable {
        Calculator calculator0 = new
            Calculator();
        double[] doubleArray0 = new double[2];
        double double0 = calculator0.sum(
            doubleArray0);
        assertEquals(0.0, double0, 0.01);

        double[] doubleArray1 = new double[2];
        doubleArray1[0] = (-1.0);
        double double1 = calculator0.sum(
            doubleArray1);
        assertEquals((-1.0), double1, 0.01);
    }
    //Test Cases for Subtract Method
    @Test(timeout = 4000)
    public void test01() throws Throwable {
        Calculator calculator0 = new
            Calculator();
        double[] doubleArray0 = new double[2];
        double double0 = calculator0.subtract(
            doubleArray0);
        assertEquals(0.0, double0, 0.01);

        double[] doubleArray1 = new double[5];
        doubleArray1[2] = 93;
        double double1 = calculator0.subtract(
            doubleArray1);
        assertEquals((-93), double1, 0.01);
    }
}
```

Listing 3: Example of test case source code for Assertion Roulette test suite.

F. Procedure

Following the results of the pilot study, we conducted two sessions: a preparation session and a controlled experiment session. During the preparation session, we supplied students with a video tutorial for both Windows and Mac to show them how to install setup the programming environment on their computers and run test cases on their IntelliJIDEA[5] panels. We also gave the students written instructions outlining a step-by-step procedure for installing Java on their systems[6]. We made sure all students had their environment ready and knew how

[4]Some details revealing the identity of the course's institution has been omitted for double-blind review.

[5]https://www.jetbrains.com/idea/

[6]These instructions are included in our replication package

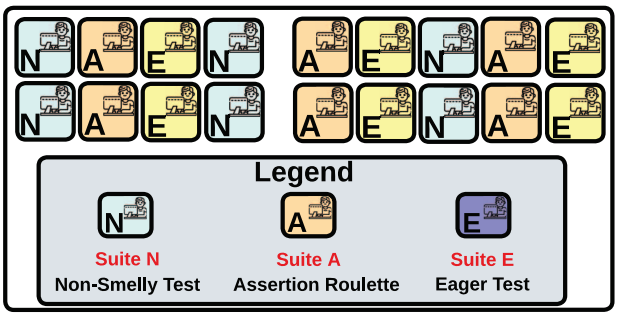

Fig. 1: Students arrangement in the classroom.

to run test cases prior to our experiment. Also, we provided students with the bug-free version of the project, along with some test cases from *Suite N*, as we want to increase student's familiarity with the production methods. Students' familiarity with the production code is important, as some students may exercise more effort to read and comprehend source code [43]. Code comprehension is also a noise that we mitigate through their exposure to the project before the session. The supporting material related to the current experiment can be accessed through the link: [1]. Finally, we conducted a presentation of the calculator project, its features (operations), its source methods, and the execution of sampled test cases.

The second session was carried out in person to avoid any collusion. At the start of the controlled experiment session, we randomly split students into three groups, based on which test they will be using: *N*, *E*, and *A*. The entire session was 2 hours (120 minutes) long. The experiment started at the same time for every student. The task assigned to the students was to identify and fix the issues raised by the failing test methods in their corresponding suite. The use of online resources was also permitted. Students were instructed to submit their updated code immediately to Canvas[7] once they are done fixing the errors. We chose to use Canvas since students are familiar with it. We determined *debugging time* for each student by examining their submission timestamp on Canvas. Finally, we shared an online post-survey to gather their feedback about their debugging experiences. We use this survey to gauge if students have experienced any difficulties when debugging their code. It is important to note that students are not aware of the underlying experiment, i.e., the multiple test suites and the existence of test smells.

G. Participants

The controlled experiment was carried out in two semesters and involved 96 undergraduate students. These participants were enrolled in a software engineering class. Students were asked to complete the two sessions to obtain extra credit, but they were given the option to choose to have their data examined as part of this study. Based on that consent, out of 56 participants from semester one and 65 from semester two, we collected data from 45 participants in the first semester and 51 participants in the second semester. Figure 1 depicts each group's final arrangement in the classroom, and Table III contains the distribution of participants in each category.

Upon the experiment's completion, there were 29 participants in Group *N* (*Non-Smelly Test*), 33 participants in Group *A* (*Assertion Roulette*), and 34 participants in Group *E* (*Eager Test*).

```
public class Calculator_ESTest extends
    Calculator_ESTest_scaffolding {
    @Test(timeout = 4000)
    public void test00()  throws Throwable {
        Calculator calculator0 = new
            Calculator();
        double[] doubleArray0 = new double[2];
        double double0 = calculator0.sum(
            doubleArray0);
        assertEquals(0.0, double0, 0.01);

        double[] doubleArray1 = new double[2];
        doubleArray1[0] = (-1.0);
        double double1 = calculator0.sum(
            doubleArray1);
        assertEquals((-1.0), double1, 0.01);

        double[] doubleArray2 = new double[2];
        double double2 = calculator0.subtract(
            doubleArray2);
        assertEquals(0.0, double2, 0.01);

        double[] doubleArray3 = new double[5];
        doubleArray3[2] = 93;
        double double3 = calculator0.subtract(
            doubleArray3);
        assertEquals((-93), double3, 0.01);
    }
}
```

Listing 4: Example of test case source code for Eager Test suite.

H. Data Collection

Data collection, in this study, is two-fold. First, we conducted a controlled experiment to determine how much time each participant incurred debugging the given source code. The participants began the lab session at the same time and submitted their corresponding source code after completing the debugging task. Second, we created a survey to gather details on participants' experiences in relation to debugging test cases. The survey questions were made available once they had finished identifying and fixing the bugs in the source code. Google Forms[8] was used to supply the participants with survey questions and collect the data. Two multiple-choice questions and one open-ended question were included in the questionnaire.

I. Survey

The initial survey had eleven questions. Then, it was revised to eliminate questions that were found repetitive, irrelevant,

[7]Web-based learning management system. https://instructure.com/canvas

[8]https://www.google.com/forms/about/

TABLE III: Summary of the number of participants in the study.

# of Semesters	# of Students	# of Eliminated Students	# of Participated Students	Categories		
				Non-Test Smell	Assertion Roulette	Eager Test
Semester One	56	11	45	14	16	15
Semester Two	65	14	51	15	17	19
Total	**121**	**25**	**96**	**29**	**33**	**34**

TABLE IV: Set of survey questions.

Question	Type
Were you able to fix all the errors detected by the test cases?	Multiple Choice
The process of finding the errors detected by the test cases was smooth and easy.	Multiple Choice
If you agreed or strongly agreed with the previous question, please explain why	Open-ended

or confusing. This revision reduced the number of questions to nine. The pilot study of the four undergraduate students revealed concerns about the length of the survey, the redundancy of some questions, and the need for logical arrangement. We reduced the nine questions to three accordingly. The final survey contains three questions—two multiple-choice and one open-ended—that can be seen in Table IV.

Survey questions were made for extra credit and only for the students who participated in debugging the code. The survey respected data privacy and protection guidelines. For instance, we protect the privacy of the respondents who participated in the study by anonymizing all responses. Additionally, passwords were used to secure the researcher's laptop with all the research materials, including participant responses. Furthermore, respondents' consent was obtained prior to participation for the utilization of their data for research purposes.

IV. Study Results

In this section, we present the impact of *Assertion Roulette* and *Eager Test* on students' debugging skills (Section IV-A) and their experience with the debugging process for each test suite (Section IV-B). Further, we discuss the results of suite test for each group *N*, *E*, and *A*.

A. Experiment Results

RQ: To what extent do *Assertion Roulette* and *Eager Test* impact the time spent by participants in debugging failing test cases?

Method. The answer to this RQ, Figure 2 reports *debugging time* boxplots of each group. To test the significance of the difference between each pair of group values, we use the Mann-Whitney U test, a non-parametric test that checks continuous or ordinal data for a significant difference between two independent groups. Our hypothesis is formulated to test whether group *A* values are significantly higher than group *N*. The difference is considered statistically significant if the p-value is less than 0.05. The same test is repeated for (group *E*, group *N*) and (group *A*, group *E*) pairs.

Results. As shown in Figure 2, group *A* values are significantly higher than the values of group *N* (*i.e.*, $p < 0.05$). Similarly, group *E* values are significantly higher than the values of group *N* (*i.e.*, $p < 0.05$). The two pairwise comparisons indicate how students who were using either Suite *A* or Suite *E* have spent a significantly larger amount of time locating the errors, in comparison with students who were using Suite *N*.

Observations. The time spent to locate an error differs depending on the test suite reporting it. For instance, when using Suite *N*, each failing test method contains only one failing assert statement. This failing statement would eventually indicate to the student the inconsistency between the expected value and the actual value of the method under test. The students would then investigate the corresponding method. When the error is found and then fixed, not only the investigated testing method would pass, but also any other test methods that were failing for the same reason. On the other hand, when using Suite *A*, each failing test method contains multiple assert statements, in which a subset is failing with more than one assert statement to be investigated. The students tend to examine them to understand the common cause for their failure before switching to the method under test to search for the error. Therefore, the investigation of the multiple asserts seems to increase the debugging time, and therefore, the *Assertion Roulette* is negatively impacting the students' debugging process. Our observation brings an alternate perspective with respect to the recent findings of Bai et al. [7], who conjectures that *Assertion Roulette* may no longer be considered a code smell. In fact, Bai et al. [7] demonstrated, through a controlled classroom experiment, that *Assertion Roulette* does impact neither the frequency of testing nor the accuracy of the test cases. We argue that *Assertion Roulette* is a program comprehension problem. Therefore, a controlled experiment where students' programming performance and testing behaviors are measured may not reveal the negative symptoms of *Assertion Roulette*. As for the *Eager Test*, the smell is caused by a test case involving multiple production methods, thereby increasing coupling between the test and production code. Such situations result in participants reviewing multiple production methods and continually switching between multiple code files as part of their troubleshooting task, which potentially increases their cognitive load and debugging time.

Summary: The presence of test smells, namely *Assertion Roulette* and *Eager Test*, in the test suite cause participants to spend more time troubleshooting test case failures, in comparison with performing the same debugging process using a non-smelly suite.

B. Survey Results

Method. We also wanted to know if the debugging task was challenging or straightforward for the participants to complete. Thus, we analyzed the participants' survey responses after they submitted them. Figure 3 presents the responses by the participants divided based on each group. We asked the participants three questions.

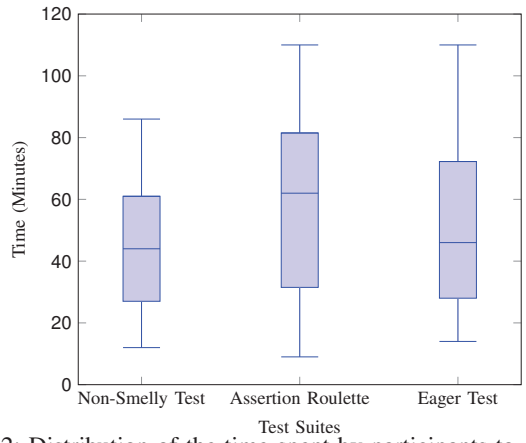

Fig. 2: Distribution of the time spent by participants to detect and debug each test suite.

Results. We questioned the participants, ***"Were you able to fix all the errors detected by the test cases?"***, to investigate the impact of test smells on the performance of participants' detection and debugging skills of the source code. As the proportion of successful participants finishing the task within the given time frame significantly impacts their performance, this question is essential to the generalizability of the analysis. We compiled the responses, and we verified them with the files that were submitted. We found that a total of 13 participants were unable to locate and fix the source code, with the number of participants assigned to the *Assertion Roulette* test suite failing by the biggest margin (n = 7). Three participants who were provided the *Non-Smelly Test* test suite did not manage to complete the assignment. Similarly, 3 participants in the test suite for the *Eager Test* failed to complete the task. These statistics illustrate the considerable detrimental impact of *Assertion Roulette* on participants' ability to debug source code. It should be highlighted that participants were uninformed of the test smells existence in the suites.

Summary: Although the participants were unaware of the test smells they were assigned, most of them managed to locate and fix all the issues in the source code. Additionally, participants assigned with the *Assertion Roulette* test suite had the highest percentage that was unable to complete the assignment within the given time frame revealing the negative impact of this particular smell.

Figure 3 showcases the response of participants regarding the second multiple choice question of the survey; ***"The process of finding the errors detected by the test cases was smooth and easy?"*** Five options—Strongly Agree, Agree, Neutral, Disagree, or Strongly Disagree—were given to the participants. The majority of participants, in general, strongly agreed that the process of identifying and debugging test cases is simple and straightforward.

Figure 3 (A) corresponds to the responses received by the participants from *Non-Smelly Test* test suite. It is clear that the highest ratio (55% n = 16) of participants agree with the smoothness and straightforwardness of the underlying debugging task. Following this, 28% (n = 8) of the participants selected the "*Agree*" option, which makes an overall 83% (n = 24) of the participants who agreed with the simplicity of locating and fixing bugs in *Non-Smelly Test* test suite. On the contrary, only a small portion of participants, i.e., 3% (n = 1), strongly disagreed hence, referring to the procedure as difficult, while none of the participants disagreed with the statement. On the other hand, 14% (n = 4) of the participants remained neutral to the asked question. Therefore, it is fair to say that the absence of test smells in the test cases makes the participants' task of debugging easy to understand and simple to fix.

Figure 3 (B) showcases the responses received by the participants of *Assertion Roulette* test suite to the survey question #2. Unlike the previously discussed case, only 67% (n = 22) participants of *Assertion Roulette* test suite coincided with the process of locating and fixing the bugs to be easy and smooth, among which 40% (n = 13) participants were in strong agreement with the statement and 27% (n = 9) of the participants agreed. Moreover, the ratio of participants who disagreed and strongly disagreed with the stated question amounted to 15% (n = 5), among which 9% (n = 3) disagreed and 6% (n = 2) of the participants strongly disagreed. These numbers may have formed due to the test cases' inclusion of numerous untitled assertions, which confused the participants. The participants spent more time locating the assertion that caused the test case to fail, making it more difficult to handle. Therefore, the most difficult test smell to address in the test cases can be ruled out as *Assertion Roulette*.

Figure 3 (C) presents the statistical data regarding participants' responses to survey question #2 in terms of *Eager Test* test suite. Overall, 82% (n = 28) of the underlying participants voted in conjunction with the stated question. Where, 50% (n = 17) strongly agreed with the ease and simplicity of the process and 32% (n = 11) of the participants responded with "*Agree*" option. On the other hand, only 3% (n = 1) of the participants disagreed with the process of locating and fixing bugs in *Eager Test* test suit being easy and smooth. In *Eager Test* test suit, the test cases from *Non-Smelly Test* and *Assertion Roulette* test suites were merged into 9 test cases. Moreover, this smell induces multiple production codes when one method is invoked, causing a significant impact on the debugging skills of the participants. However, participants found it comparatively easy to handle as they were able to follow up on the logic of the operation and locate the bug in the source code.

Observations. Overall, students from all groups exhibit similar levels of satisfaction with their testing experience, with a slight increase for those using non-smelly tests, and those using *Eager Tests*. None of the participants have reported any issues they experienced, despite the smelliness of Suite *A* and *E*. We argue that *Assertion Roulette* and *Eager Test* are hard to be sought as problematic, as they are intuitive by nature. This is aligned with their high frequency in open source systems,

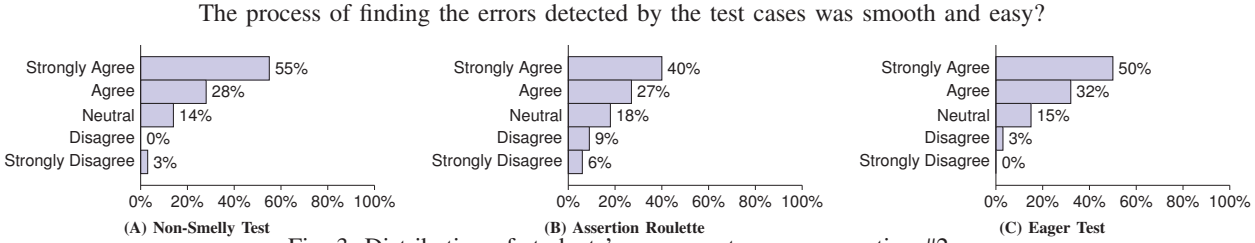

Fig. 3: Distribution of students' responses to survey question #2.

according to recent studies [10], [24], [33], [39].

Summary: *Non-Smelly Test* and *Eager Test* test suites were equally rated to be simple and straightforward to handle. In contrast, *Assertion Roulette* test smell was rendered to make the process difficult for the participants.

Finally, we supplied the participants with an open-ended question: ***If you agreed or strongly agreed with the previous question, please explain why.*** Responses received from group *N* participants show their high satisfaction with the overall testing experience:

> 💬 ***Comment 1:*** *"The structure was easier to diagnose with each test cases. Once I was able to diagnose where the errors were, it got easier to clear all the errors."*

> 💬 ***Comment 2:*** *"Once I found which test case was which, it was easy to find what function they were using (after I realized the equate function was a built in one). That in turn made it easy to figure out that the error was limited in scope to that function alone [...]"*

Moreover, some participants who were not previously familiar with the Java programming language expressed a positive experience, as one of the participants said:

> 💬 ***Comment 3:*** *"Haven't touched java before, once I understood what was going on, wasn't difficult to find"*

Responses received from group *A* participants did not differ from the previous group, as students expressed their satisfaction with the debugging process:

> 💬 ***Comment 4:*** *" [...] Because I can see the expected result and the actual result and trace back where the code is wrong."*

Responses received from group *E* participants did not deviate from the previous ones, as students positively described their debugging:

> 💬 ***Comment 5:*** *"The print out statement and the expected output helped me identified where in the code to look and what I needed to fix."*

> 💬 ***Comment 6:*** *"The test cases pointed out some functions being used and gave the expected and actual values. This helped me to pinpoint what was wrong and where."*

> 💬 ***Comment 7:*** *"I found it easy to to find the issues because the code was well formatted and commented. When I saw the error was with subtraction I went to the subtraction function. There I read the comments on what each part of the code was supposed to do, when it was different I just changed the code to do what the comments said."*

Among these positive comments, We note how *comment 6* has mentioned the test of more than one method without necessarily considering it to be problematic. It is apparent that the successful fix of all errors raised a sense of accomplishment among students and positively influenced them.

Summary: *Assertion Roulette*, and *Eager Test* smells are transparent to students as long as they are successful in locating and fixing errors.

V. Discussion

The analysis of the current investigation has revealed significant information regarding the impact of *Assertion Roulette* and *Eager Test* on the participants' debugging time.

As we previously discussed, Bai et al. [7] advocates for *Assertion Roulette* to be no longer considered as test smell. However, the aforesaid statement is not consistent with our findings. Our experiment revealed that participants spent significantly more time sorting the assertions that are stacked in the *Assertion Roulette* test methods. Furthermore, we witnessed a similar pattern of longer debugging time, when students use *Eager Test* test methods to locate errors.

Therefore, educators need to raise the students' awareness of writing smell-free test cases. Students need to be taught how to avoid writing multiple asserts under the same test methods, or to test multiple production methods, using the same test method. In this context, Buffardi et al. examined students' test methods on their assignments, and indicated potential problems in their unit tests. In fact, the top three common patterns detected in their test methods: multiple member function calls, multiple assertions, and conditional logic [13]. Thereby, taking an action to educate students on how to avoid these anti-patterns, would decrease the early propagation of these smells.

Additionally, the current research discovered that test cases with descriptive names and commented asserts increase their readability [37]. Thus, we encourage students to develop the practice of documenting their test methods.

A. Lessons Learned:

Throughout the experiment, several lessons were learned, and perceptual observations were conducted.

Lesson #1: In addition to teaching students about design and code smells, academia must also instill in students the importance of writing quality test cases, specifically test smells and the harm caused by the introduction and existence of test smells in the systems code base. Furthermore, teaching students about code reviews should not be limited to the production code but should also include the test suites, as such code is vital to the system's overall quality.

Lesson #2: The research community has produced tools to aid developers with detecting (and, in some cases) correcting test smells for various programming languages, and testing frameworks [3]. These tools have been utilized in multiple empirical studies and have been effective in their detection mechanism. Academia should promote using these tools in the classroom to better equip students with means of evaluating the quality of the test cases they produce for class assignments. In addition, the automatic detection of test smells will help students to troubleshoot issues much faster.

Lesson #3: Even though most research on test smells focuses on Java systems, test smells are not unique to JUnit-based test suites. Therefore, the research and academic community should invest in exploring the types of test smells that are unique to specific programming languages and paradigms and passing that knowledge to students in the classroom so that they are better prepared when entering the workforce.

VI. Threats to Validity

The applicability of the findings in this research is susceptible to a number of threats.

Internal Validity. The students who participated in this experiment were from the same university, making the participant pool relatively coherent. To mitigate this, we run our experiments in two semesters. Another pertinent threat relates to the choice of the project under test. The complexity of the project features may require advanced debugging skills, which may crate a significant overhead in our experiment. To mitigate this issue, we developed a calculator application with eight operations. The choice of the calculator ensures its intuitiveness to the students. Although some students might not be familiar with the source code, we gave them detailed instructions and documentation to familiarize themselves with the project and source code in order to minimize the threat and potential misunderstandings.

External Validity. This study is exclusively focused on two test smells. However, given the prevalence of test smells, there may be a chance that we missed an important smell that is typically written by developers. To mitigate this threat, we reviewed various research papers that have conducted similar types of experiences. We concluded that the most popular and frequent smells are *Assertion Roulette*, and *Eager Test*. Moreover, it is essential to replicate the study with a broader and more varied range of test smells along with a wider range of codebases.

VII. Conclusion and Future Work

In this study, we explored the impact of test smells on a student's program comprehension ability. The results indicate that the participants in our study who were assigned test suites containing test smells took more time to complete the assigned task of fixing errors in the production code than those given test suites without test smells. Furthermore, the smell *Assertion Roulette* had a greater impact on troubleshooting and debugging than the *Eager Test* smell. For future work, we plan to conduct a similar study with additional test smell types to determine how these new smells impact a student's ability to comprehend code when performing maintenance activities.

Verifiability and Replicability

To enable full verifiability and replicability, our experimental materials are available[9] [1].

References

[1] Supplementary materials. https://wajdialjedaani.github.io/testsmellstd/.

[2] K. Aaltonen, P. Ihantola, and O. Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 153–160, 2010.

[3] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C. D. Newman, A. Ghallab, and S. Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[4] M. Aniche, F. Hermans, and A. Van Deursen. Pragmatic software testing education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 414–420, 2019.

[5] P. Baheti, L. Williams, E. Gehringer, and D. Stotts. Exploring pair programming in distributed object-oriented team projects. In *Educator's Workshop, OOPSLA*, pages 4–8. Citeseer, 2002.

[6] G. R. Bai, B. Clee, N. Shrestha, C. Chapman, C. Wright, and K. T. Stolee. Exploring tools and strategies used during regular expression composition tasks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 197–208. IEEE, 2019.

[7] G. R. Bai, K. Presler-Marshall, S. R. Fisk, and K. T. Stolee. Is assertion roulette still a test smell? an experiment from the perspective of testing education. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–7. IEEE, 2022.

[8] G. R. Bai, K. Presler-Marshall, T. W. Price, and K. T. Stolee. Check it off: Exploring the impact of a checklist intervention on the quality of student-authored unit tests. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, pages 276–282, 2022.

[9] G. R. Bai, J. Smith, and K. T. Stolee. How students unit test: Perceptions, practices, and pitfalls. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, pages 248–254, 2021.

[10] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. 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)*, pages 56–65. IEEE, 2012.

[11] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[12] K. Beck. *Test-driven Development: By Example*. Addison-Wesley signature series. Addison-Wesley, 2003.

[13] K. Buffardi and J. Aguirre-Ayala. Unit test smells and accuracy of software engineering student test suites. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, pages 234–240, 2021.

[9] https://wajdialjedaani.github.io/testsmellstd/

[14] B. Camara, M. Silva, A. Endo, and S. Vergilio. On the use of test smells for prediction of flaky tests. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '21, page 46–54, New York, NY, USA, 2021. Association for Computing Machinery.

[15] J. C. Carver and N. A. Kraft. Evaluating the testing ability of senior-level computer science students. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pages 169–178. IEEE, 2011.

[16] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing (JERIC)*, 3(3):1–es, 2003.

[17] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30, 2004.

[18] S. H. Edwards and Z. Shams. Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 171–176, 2014.

[19] S. H. Edwards, Z. Shams, M. Cogswell, and R. C. Senkbeil. Running students' software tests against each others' code: new life for an old" gimmick". In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 221–226, 2012.

[20] G. Fraser, A. Gambi, M. Kreis, and J. M. Rojas. Gamifying a software testing course with code defenders. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 571–577, 2019.

[21] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, 138:52–81, 2018.

[22] A. Gaspar, S. Langevin, N. Boyer, and R. Tindell. A preliminary review of undergraduate programming students' perspectives on writing tests, working with others, & using peer testing. In *Proceedings of the 14th annual ACM SIGITE conference on Information technology education*, pages 109–114, 2013.

[23] M. H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. *ACM SIGCSE Bulletin*, 34(1):271–275, 2002.

[24] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, 2019.

[25] P. Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Series. O'Reilly Media, 2004.

[26] J. In. Introduction of a pilot study. *Korean journal of anesthesiology*, 70(6):601–605, 2017.

[27] A. M. Kazerouni, S. H. Edwards, and C. A. Shaffer. Quantifying incremental development practices and their relationship to procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 191–199, 2017.

[28] A. M. Kazerouni, C. A. Shaffer, S. H. Edwards, and F. Servant. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 407–413, 2019.

[29] V. Khorikov. *Unit Testing Principles, Practices, and Patterns*. Manning, 2020.

[30] D. J. Kim, T.-H. P. Chen, and J. Yang. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26(5):1–47, 2021.

[31] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[32] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn. Test smells 20 years later: Detectability, validity, and reliability. *Empirical Software Engineering*, 2022.

[33] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. 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, page 193–202, USA, 2019. IBM Corp.

[34] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[35] R. Pressman and B. Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2019.

[36] J. Robergé and C. Suriano. Using laboratories to teach software engineering principles in the introductory computer science curriculum. *ACM SIGCSE Bulletin*, 26(1):106–110, 1994.

[37] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[38] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12, 2018.

[39] D. Spadini, M. Schvarcbacher, A.-M. Oprescu, M. Bruntink, and A. Bacchelli. Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 311–321, New York, NY, USA, 2020. Association for Computing Machinery.

[40] B. van Bladel and S. Demeyer. Test refactoring: a research agenda. In *CEUR workshop proceedings*, volume 2070, pages 1–6, 2017.

[41] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, 2001.

[42] L. Williams and R. L. Upchurch. In support of student pair-programming. *ACM SIGCSE Bulletin*, 33(1):327–331, 2001.

[43] L. Yenigalla, V. Sinha, B. Sharif, and M. Crosby. How novices read source code in introductory courses on programming: an eye-tracking experiment. In *International Conference on Augmented Cognition*, pages 120–131. Springer, 2016.