Is Assertion Roulette still a test smell? An experiment from the perspective of testing education

Gina R. Bai∗, Kai Presler-Marshall∗, Susan R. Fisk† and Kathryn T. Stolee∗
∗Department of Computer Science, North Carolina State University
†Department of Sociology, Kent State University
rbai2@ncsu.edu, kpresle@alumni.ncsu.edu, sfisk@kent.edu, ktstolee@ncsu.edu

Abstract—Test smells are commonly perceived as having a negative impact on software maintainability and correctness. Research has shown that Assertion Roulette is the most pervasive smell in industrial and open-source systems. However, some recent studies argue that the impact of Assertion Roulette is not as severe as previously believed, and developers usually consider it acceptable.

The controversy over the impact of Assertion Roulette also exists in the area of testing education. To assess the impact of Assertion Roulette, we conducted a controlled empirical study with 42 CS students. We recruited participants from two populations, CS1 and a graduate testing course, to see what role experience may have in terms of this test smell’s impact. Participants were tasked with implementing a project in Java that passes provided JUnit tests. Through analysis of student-authored source code, we measured the impact of Assertion Roulette using code quality measures and testing behavior measures. Our findings show that the impact of Assertion Roulette on students in this study was minimal. Though students with exposure to the test smell began testing significantly later, they performed similarly in terms of programming quality measures. Thus, it would seem the Assertion Roulette smell is no longer a smell at all, even for less experienced populations like students.

Index Terms—test smell, assertion roulette, computer science education, testing education

I. INTRODUCTION

Smells in code can be frustrating, and they are not limited to source code. Smells also exist in test code [1]. The test smell Assertion Roulette was formally defined in 2001 [1], and it occurs when a test case contains multiple assertions without documentation/explanation in case one of them fails. Prior work found it to be the most diffused test smell [2], [3], and that it frequently co-occurs with simple bugs [4] and other test smells [2], [5], [6]. Assertion Roulette is also notorious for its negative impacts on the maintainability [2], [7]–[9], readability [3], [6], [8], and correctness [2], [10] of both source code and test code. This smell can also lead to a high rate of code churn [11], [12].

However, recent studies argue that this smell may not actually be a smell [13] in that developers are not bothered by it. Developers rarely refactor this smell in GitHub commits [10], and commonly perceive the smell as low severity [14]–[16], and hence view Assertion Roulette as an “acceptable trade-off between maintainability and ease of writing” [14]. In fact, having this smell may actually indicate test robustness and increase the quality of production code [17].

The understanding of Assertion Roulette also varies among educators. Bavota et al. [2] reported that the presence of Assertion Roulette significantly reduced CS1 students’ correctness in a program maintenance task, specifically, students in their study were less successful at manually identifying which line of code in the given test suite generated a particular error trace and at understanding what had gone wrong based on the error trace. Buffardi et al. [18] explored the relationship between the existence of test smells, such as Conditional Logic and Assertion Roulette, in a student-authored test suite and test suite accuracy. Yet, they found no relationship between the accuracy of student-authored tests and the presence of Assertion Roulette.

In this work, we conduct an experimental, controlled study to evaluate the impact of the Assertion Roulette smell on students’ performance and behaviors during source code composition. We recruited students early in their CS education, namely CS1, and later, in a graduate course, to see what role experience may have with respect to the test smell’s impact.

We frame this study around two research questions:

RQ1: How does Assertion Roulette in the provided test suite impact students’ programming performance?

RQ2: How does Assertion Roulette in the provided test suite impact students’ testing behaviors?

In this study, 42 students participated in a two-hour lab activity and implemented source code for a Java-based project. We stratified the students based on education (CS1 vs. graduate), and then randomly assigned them into experimental and control groups. The experimental group received the test suite with Assertion Roulette (AR group) and the control group received test suite without this smell (AR-free group). We then analyze student-authored source code, and reveal the impact of Assertion Roulette on students’ programming performance and testing behaviors. Our results show that:

• The impact of the presence of Assertion Roulette in the provided test suite is minimal, but

• The presence of Assertion Roulette significantly impacted when students began testing; students faced with Assertion Roulette began testing later.

This work contributes to research on testing education. It is the first study on the impact of Assertion Roulette on students’ programming performance and testing behaviors.
II. BACKGROUND AND RELATED WORK

In this section, we illustrate the Assertion Roulette smell using the implementation task we used in this study (Section II-A). We also discuss prior work on the assessment of students’ programming and testing activities (Section II-B).

A. Assertion Roulette

In this study, students were tasked with implementing a Java program Bowling Score Keeper (BSK) to score a bowling game. We chose this project as it has been used in numerous other testing-related studies (e.g., [19]–[24]).

Assertion Roulette comes from “having a number of assertions in a test method that have no explanation” [1]. Hence, to assess the impact of Assertion Roulette, we intentionally seeded this smell into the test code for the experimental group (AR group) by combining all test methods for each user story into a single test method (seven in total, details in Section III-B). For example, one of user stories in BSK was:

```
public class US01 {
    @Test
    public void testEmptyFrameIsCreated()
        final Frame f = new Frame();
        assertNotNull(f);
}
```

**User Story 1 - Frame:**
Each turn of a bowling game is called a frame. Ten pins are arranged in each frame. The goal of the player is to knock down as many pins as possible in each frame. The player has two chances, or throws, to do so. The value of a throw is given by the number of pins knocked down in that throw.

The AR-free (control) version of the test class (Figure 1) contained test cases that each had a descriptive name that reflected the expected program behavior [14] and a single assertion [25]. For example, the test case `testFrameWithScoreIsCreated()` verifies that a frame consists of two chances/throws, and the test case `testExceptionMoreThan10PinsPerFrame()` verifies that an exception is thrown when trying to create a Frame with more than ten pins knocked down. Figure 2 shows the Assertion Roulette version of the test suite for User Story 1, with all assertions in one test case (lines 5, 8, and 19). We rewrote the test `testExceptionMoreThan10PinsPerFrame()` (lines 14-18 in Figure 1) with try-catch block (lines 10-16 in Figure 2), which ensures that an Exception was thrown in the right place in the test case `test()`. The presence of Assertion Roulette results in fewer test cases and less granular test results. In JUnit, a test case will stop executing when an assertion fails or when an Exception is thrown. This means that, for example, a student with a test containing Assertion Roulette (e.g., Figure 2) who has a fault revealed by the final assertion would see that they are passing 0 of 1 test cases with a pass rate of 0%. That same student with the same fault and a test suite absent of Assertion Roulette (e.g., Figure 1) would pass 3 of 4 test cases with a pass rate of 75%. In this way, the lack of Assertion Roulette presents more positive or encouraging feedback.

Unlike prior work [18] that focuses on the quality of student-authored tests, we consider how Assertion Roulette in a provided test suite impacts students’ development efforts.

```
public class US01 {
    @Test
    public void testEmptyFrameIsCreated(){
        final Frame f = new Frame();
        assertNotNull(f);
    }
    @Test
    public void testFrameWithScoreIsCreated(){
        final Frame f = new Frame(1, 2);
        assertNotNull(f);
    }
    @Test { expected = Exception.class }
    public void testExceptionMoreThan10PinsPerFrame(){
        final Frame f = new Frame(12, 12);
        assertNotNull(f);
    }
    @Test
    public void testFrameIsCreatedWithCorrectName(){
        final Frame f = new Frame();
        assertEquals("Frame",
            f.getClass().getSimpleName());
    }
    @Test
    public void testFrameIsCreatedWithCorrectNameWithException(){
        final Frame f = new Frame();
        assertEquals("Frame",
            f.getClass().getSimpleName());
    }
}
```

**Fig. 1:** Example test class that lacks Assertion Roulette

```
public class US01 {
    @Test
    public void test()
        Frame f = new Frame();
        assertNotNull(f);
        f = new Frame(12, 12);
        fail();
        catch (final Exception e){
            // Exception expected
        }
        f = new Frame();
        assertEquals("Frame",
            f.getClass().getSimpleName());
}
```

**Fig. 2:** Example test class that exhibits Assertion Roulette

B. Students’ programming and testing activities

Researchers and educators often measure the accuracy of student-authored source code via the number of passed test cases/suites [26]–[28] while students’ productivity is often evaluated using the number of lines of code (LOC) changed per hour [29] or per work session [26]. Pettit et al. [30] investigated how students modify their programs given automated assessment tools, using more than 45,000 CS1 student submissions collected over seven semesters. They evaluated students’ programming progress using the number of LOC changed per hour or per work session [26]. Pettit et al. [30] investigated how students modify their programs given automated assessment tools, using more than 45,000 CS1 student submissions collected over seven semesters. They evaluated students’ programming progress using the number of LOC changed per hour or per work session [26].

In this study, we quantify students’ programming performance in terms of: 1) accuracy, such as how many assertions...
and test cases does student-authored source code pass, and 2) effectiveness, for example, the number of LOC changed between two consecutive test suite runs, and the time intervals between test suite runs. We also investigate students’ testing behaviors by studying the number of test suite executions, how early students start testing, and how often they test.

III. STUDY

In this section, we describe the study design and execution; materials are available on GitHub [34].

A. Procedure

This study was conducted over six two-hour lab sessions held synchronously via Zoom. Each student attended one lab session. At the beginning of the lab session, students received a 20-minute introduction on the procedure of the study, an overview of the programming project, and a step-by-step live demo on project setup. Students were guided to a GitHub repository containing links to the video version of the setup instructions and the project. Students were allowed to consult any online resources for assistance throughout the study.

B. Implementation Task

1) Project: Students implemented the Java-based project BSK, given requirements and a test suite as a starting point. Prior to the study, we conducted a pilot study with five CS undergraduate and graduate students to clarify the programming project and to ascertain the potential duration of the lab session. As a result, we shortened the original BSK project, which consists of 13 user stories, down to the first seven so it could be completed in 90 minutes by students. Our shortened version of the BSK project contains 37 lines of Java source code (skeletons for the three necessary classes) and 558 lines of Java test code (JUnit tests). Each user story had a separate test class, with three to sixteen test cases per user story.

We used an experimental, controlled study design, wherein we randomly assigned students, stratifying based on education, to a control group (AR-free group) or an experimental group (AR group). Both groups received the same project requirements. What differed between the groups was the number of test cases and the number of assertions per test case. Students in the AR-free group worked on the BSK project with tests that lack Assertion Roulette in the test suite, while students in the AR group worked on the BSK project with test suites that exhibited Assertion Roulette. To verify the equivalence of the two test suites, we used mutation testing, which serves as an approach to compare the bug-revealing capabilities of two different test suites. In particular, we confirmed there were no differences in the bug-revealing capability of both test suites via PITest [35], a commonly-adopted Java mutation testing tool. We considered both versions of the test suites equivalent as they killed the same set of mutants.

2) Development Environment: To ensure a consistent development environment, we assigned each student a virtual machine image (VM) with a standard Linux development environment: Ubuntu 18.04 and Eclipse Java 2020-06, Nano, Vim, and Gedit.

The VMs were set up with a script for students to run the test suite on their project implementation using Maven’s test runner. After every run of the test suite, the script would display test results to the student and make a timestamped copy of the project, including all code and the test results. Instructions for using the VM and running the script were included on both the VMs and the GitHub repository, and one of the authors gave a live demonstration in each lab session. While the use of a script for building the projects and running the tests is a departure from the built-in Eclipse JUnit interface, it allowed us to take a snapshot of the entire project each time the test suite was run in order to study the evolution of the code and the types of changes made. To reduce the barrier from our testing script, we integrated the script with the Eclipse debugger so that students could debug the tests interactively. An example of the test results that students saw is shown in Figure 3. The script reported the number of tests run, the number of tests that failed (due to a failing assertion or unexpected exception), the overall status of the build, and detailed results on each test that failed.

C. Participants

We conducted this study with 49 students: 24 graduate students and 25 undergraduate students from North Carolina State University. The graduate students were recruited from a graduate-level software testing course. All students in this course were required to complete this study as an in-class workshop to receive credit for the class activity, but were allowed to opt out of having their data analyzed for this study. The undergraduate students were recruited from a CS1-level Java programming course and were eligible to receive extra credit upon completion of the study.

1 The VMs were supported and managed by NCSU’s virtual computing lab.
TABLE I: ANOVA results for comparing the AR group and the AR-free group

<table>
<thead>
<tr>
<th>Dependent Variable</th>
<th>Independent Variable</th>
<th>isAR</th>
<th>isCS1</th>
<th>isAR</th>
<th>isCS1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) totalRun</td>
<td></td>
<td>0.6420</td>
<td>0.3630</td>
<td>0.7000</td>
<td></td>
</tr>
<tr>
<td>2) testFirst</td>
<td></td>
<td>0.0282 *</td>
<td>0.0058 **</td>
<td>0.0714</td>
<td></td>
</tr>
<tr>
<td>3) testFreq</td>
<td></td>
<td>0.1957</td>
<td>0.0726</td>
<td>0.0662</td>
<td></td>
</tr>
</tbody>
</table>

*p < 0.05, ** p < 0.01, *** p < 0.001

After discarding data from students who did not follow the instructions, we were left with data from the remaining 42 students for analysis. Overall, there were 18 students in the AR group (11 CS1s, 7 grads), and 24 students in the AR-free group (12 CS1s, 12 grads).

D. Data and Analysis

We collected students’ timestamped implementations from their assigned VMs upon completion of the study. In total, 42 students generated 537 timestamped copies of the project (AR group: 220 copies, AR-free group: 317 copies) by running the test suite. The projects were anonymized and the group condition was concealed during data analysis.

We measured students’ programming performance in terms of accuracy and effectiveness:

- **Accuracy**: We adapted the metrics from prior work, and we evaluated the accuracy via pass rate at both assertion (49 in total, passAstn) and user story (7 in total, passUS) levels [26]–[28], and the improvements that students made between two consecutive test suite runs in terms of assertion-level pass rate (passDelta) [28].

- **Effectiveness**: The effectiveness of students’ programming activities was measured with LOC deleted from (LOCdel) or added to (LOCadd) the project between two consecutive test suite runs [30], the total time spent on the project (duration), and the time interval between two consecutive test suite runs (runInt) [26], [31].

We adapted metrics from prior work to measure students’ testing behaviors in terms of testing effort and process:

- **Testing Effort**: We report students’ testing effort with the total number of test suite runs (totalRun) [28].

- **Testing Process**: We quantify students’ testing process via how early they start testing (testFirst) [26], [33], and how often they test (testFreq) [26].

Treating each of the 10 metrics as a dependent variable, we measured the impact of the following independent variables: isAR for comparing the AR group to the AR-free group, and isCS1 for the level in school. We used a two-way ANOVA analysis to explore the impact of each variable independently as well as their interaction (isAR * isCS1). We report the p-values in Table I.

TABLE II: Measuring students’ programming performance (i.e., accuracy and effectiveness) and testing behaviors (i.e., testing effort and testing process)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Group</th>
<th>Overall</th>
<th>CS1</th>
<th>Grad</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>med</td>
<td>avg</td>
<td>med</td>
</tr>
<tr>
<td>passAstn</td>
<td>AR</td>
<td>71.1</td>
<td>90.8</td>
<td>56.0</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>78.7</td>
<td>91.8</td>
<td>65.7</td>
</tr>
<tr>
<td>passUS</td>
<td>AR</td>
<td>68.6</td>
<td>85.7</td>
<td>57.1</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>75.7</td>
<td>85.7</td>
<td>61.4</td>
</tr>
<tr>
<td>duration</td>
<td>AR</td>
<td>71.4</td>
<td>68.0</td>
<td>76.9</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>69.1</td>
<td>70.3</td>
<td>71.9</td>
</tr>
<tr>
<td>LOCdel</td>
<td>AR</td>
<td>1.9</td>
<td>1.0</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>1.7</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>LOCadd</td>
<td>AR</td>
<td>5.5</td>
<td>2.0</td>
<td>5.8</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>4.2</td>
<td>2.0</td>
<td>4.7</td>
</tr>
<tr>
<td>runInt</td>
<td>AR</td>
<td>6.4</td>
<td>3.3</td>
<td>7.6</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>5.7</td>
<td>3.4</td>
<td>6.2</td>
</tr>
<tr>
<td>totalRun</td>
<td>AR</td>
<td>12.2</td>
<td>12.5</td>
<td>11.1</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>13.2</td>
<td>11.5</td>
<td>12.6</td>
</tr>
<tr>
<td>testFirst</td>
<td>AR</td>
<td>18.7</td>
<td>12.4</td>
<td>26.9</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>9.4</td>
<td>6.9</td>
<td>12.2</td>
</tr>
<tr>
<td>testFreq</td>
<td>AR</td>
<td>8.3</td>
<td>5.3</td>
<td>10.5</td>
</tr>
<tr>
<td></td>
<td>AR-f</td>
<td>6.4</td>
<td>5.5</td>
<td>6.6</td>
</tr>
</tbody>
</table>

IV. Results

In this section, we explore the impact of Assertion Roulette on students’ programming performance (Section IV-A) and their testing behaviors (Section IV-B).

A. RQ1: Programming Performance

Table II shows the accuracy metrics of student-authored source code, including the percentage of passed assertions and fully passed user stories (passAstn, passUS), and the progress in terms of the improved assertion-level pass rate between two consecutive test suite runs (passDelta). Table II also contains the metrics that reflect effectiveness: the time that students spent on this project (duration), the number of LOC changed in two consecutive test suite runs (LOCdel, LOCadd), and how often students ran the tests (runInt). For example, on average, CS1 students in the AR group were able to pass 56.0% of assertions in 76.9 minutes, and during development they ran the project every 3.3 minutes and passed 2.5 more assertions each time (passDelta of 5.2%) by deleting 1.8 lines of code and adding 5.8 lines of code to the project. Overall, they fully passed the test cases for four user stories out of seven (57.1%). Across the entirety of the project, students added an average of 28.4 lines of code to the project.

We found Assertion Roulette had no statistically significant impact on all programming performance measures (Variables 1-7 in Table I). These results indicate students in the AR-free group achieved no higher pass rates and programmed no more effectively than those in the AR group. Unsurprisingly, graduate students significantly outperformed the CS1 students in terms of accuracy (Variables 1-3 in Table I) and their programming performance, which gave them more experience in programming and testing (graduate students were recruited from a software testing course).
Summary: The presence of Assertion Roulette in the provided test suite has no statistically significant impact on students’ programming accuracy and effectiveness.

B. RQ2: Testing Behaviors

We present the metrics used for analyzing the testing behaviors in Table II, including the number of test suite runs (totalRun), when students started testing (testFirst), and how often they tested (testFreq). For example, on average, students in the AR group tested every 8.3 minutes and they tested 12.2 times in total, while the students in the AR-free group ran the test suite every 6.4 minutes and they tested 13.2 times in total. Neither difference was significant.

Table II shows that students in the AR-free group started testing their source code in 9.4 minutes on average, and students in the AR group started testing after 18.7 minutes. The difference was statistically significant ($p = 0.03$ for testFirst, Table I). Along with the observation in Section IV-A that students in the AR group achieved slightly lower pass rates, this result supports previous findings [26] that students who test earlier are more likely to have higher-quality implementations. We saw the greatest impact here on CS1 students (26.9 min vs. 12.2 min). It is possible that these students, who are less familiar with how to break up a task into smaller pieces, erroneously aimed for passing an entire test case at a time. Hence, having few large test cases may have discouraged novices from testing early.

Summary: Having Assertion Roulette had no statistically significant impact on how often students test, but it had a statistically significant negative impact on how early students start testing.

V. DISCUSSION

This study shed light on the impacts of Assertion Roulette on students’ programming performance and testing behaviors.

A. Is Assertion Roulette a problem for students?

In our study, both CS1 and graduate students were largely unimpeded by this smell. This echoes the findings in recent work that the impact of Assertion Roulette has diminished [13]–[16]. Along with the findings in recent prior work [18] that the presence of Assertion Roulette in student-authored test code did not impact test accuracy, we conjecture that Assertion Roulette may no longer smell bad to students. We suggest replication studies with more diverse sets of participants, and projects in different sizes and programming languages.

B. Is Assertion Roulette acceptable in testing education?

We found that the impact of Assertion Roulette on students in this study was minimal. However, we consider that it is still a good practice for instructors to avoid this smell when introducing and providing test cases to students. Having fewer but well-documented assertions in each test case can improve its readability [36], [37], and hence leads to better programming and debugging outcomes for novices [38]. Additionally, we observed that Assertion Roulette had a statistically significant impact on the time at which students start testing during implementation. This could negatively impact their source code quality [26], and might become more substantial as the projects become more complex.

Moreover, as better testing practices could lead to better programming outcomes for students [39], we encourage novice programmers and testers to document each test case; for example, with explanatory error messages and descriptive test names [14]. We also suggest that instructors discuss the pros and cons of different refactoring strategies for Assertion Roulette. For example, a commonly adopted alternative approach to get rid of this smell is to split up tests into single-assertion tests [25], [40]. However, this strategy results in a larger set of tests, which might be acceptable at the scale of student assignments or projects, but it could negatively impact test suite runtime when scaled to large applications, or in the context of regression testing or continuous integration environments [41].

VI. THREATS TO VALIDITY

Conclusion: The statistically insignificant results might result from the small sample size ($n = 42$) [42].

Construct: Students were required to work on study tasks individually and remotely in a limited time, which could potentially impact their behaviors. Additionally, students were recruited with two different incentives—graduates participated as a required class activity, while the CS1 students self-selected into this study and participated for extra credit. A replication with a more consistent participant recruitment, and a more diverse and larger set of students is needed.

Internal: While students were given the option to complete the TDD activity within the Eclipse IDE, they were required to compile and run the project with a provided script via the command line outside of Eclipse. Compared to the Eclipse JUnit runner, this may impact students’ programming performance and particularly testing behavior. However, we discarded data from students who did not follow the instructions.

External: The conclusions were drawn based on students’ performance and behaviors when using Java and JUnit, which may not generalize to other languages.

VII. CONCLUSION AND FUTURE WORK

We studied how Assertion Roulette impacts students’ programming performance and testing behaviors. Though the presence of Assertion Roulette had no statistically significant impact on how often students test, it had a statistically significant impact on when they started testing the code. Additionally, we found that the presence of Assertion Roulette had no statistically significant impact on the accuracy of student-authored source code. Hence, we conjecture that Assertion Roulette may no longer smell bad to students. For future work, a replication study with a more diverse and larger set of students is suggested to better generalize how Assertion Roulette impacts testing education.

