# Exploring the Bug Investigation Techniques of Intermediate Student Programmers

Rifat Sabbir Mansur
rifatsm@vt.edu
Virginia Tech
Blacksburg, Virginia

Ayaan M. Kazerouni*
ayaank@calpoly.edu
California Polytechnic State University
San Luis Obispo, California

Stephen H. Edwards
edwards@cs.vt.edu
Virginia Tech
Blacksburg, Virginia

Clifford A. Shaffer
shaffer@vt.edu
Virginia Tech
Blacksburg, Virginia

## ABSTRACT

Bug investigation – testing and debugging – is a significant part of software development. Ineffective practices of bug investigation can greatly hinder project development. Therefore, we seek to understand bug investigation practices among intermediate student programmers. To this end, we used a mixed-methods approach to study the testing and debugging practices of students in a junior-level Data Structures course with 3–4 week long projects. First, we interviewed 12 students of varying project performances. From the interviews, we identified five techniques that students use for both testing and debugging: 1) writing diagnostic print statements, 2) unit testing, 3) using source-level debugger, 4) submission to online auto-grader, and 5) manual tracing. Using the Grounded Theory approach, we developed four hypotheses regarding students' use of multiple techniques and their possible impact on performance. We used clickstream data to analyze the level of use of the first four of these techniques. We found that over 92%, 87%, and 73% of the students used JUnit testing, diagnostic print statements, and the source-level debugger, respectively. We found that the majority of the students (91%) used more than one technique to investigate bugs in their projects. Moreover, students who used multiple techniques had overall better performance in the projects. Finally, we identified some ineffective practices correlated with lower project scores. We believe that the findings of our research will help understand, characterize, and teach better practices of bug investigation.

## CCS CONCEPTS

• **Social and professional topics** → *Computing education*; • **Software and its engineering** → *Software development process management*; **Software testing and debugging**.

*Some work performed while at Virginia Tech.


*Koli Calling '20, November 19–22, 2020, Koli, Finland*

ACM ISBN 978-1-4503-8921-1/20/11…$15.00
https://doi.org/10.1145/3428029.3428040

## KEYWORDS

CS education, bug investigation, testing, debugging, post-CS2, data structures and algorithms

**ACM Reference Format:**
Rifat Sabbir Mansur, Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2020. Exploring the Bug Investigation Techniques of Intermediate Student Programmers. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling '20), November 19–22, 2020, Koli, Finland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3428029.3428040

## 1 INTRODUCTION

Debugging, testing, and verification are essential steps in software development. These three steps can take up to 75% of the total development cost [12]. Imperfect debugging and testing practice affects the total time spent on a given software development effort [22]. Therefore, it is essential for computer science (CS) students to learn the best practices for debugging, testing, and verification. This is even more true when developing large and complex software projects. Most CS students learn about these software development practices as early as the CS1 and CS2 courses. However, some students struggle with debugging and testing even after successfully completing CS1- and CS2-level courses. These imperfect practices in debugging and testing can greatly hinder the performance of the students in later CS courses, and in their professional life. Results of these difficulties may be manifesting in the post-CS2 Data Structures course at our university, where 25–30% of students drop out or achieve lower than a C grade each semester. In our research, we aim to explore the different techniques used by student programmers for finding and fixing bugs in a post-CS2 Data Structures and Algorithms course (what we refer to as "CS3").

Most studies on student testing and debugging practices are directed toward students who are just learning to program, in their first or second programming courses. For our research, we are interested in the practices of more advanced students. These students have passed their first year programming courses and are attempting to write and debug programs that are longer and have more complex structure and interactions. We focus on a junior-level Data Structures and Algorithms course. These students must complete four programming assignments written in Java with lifecycles of 3–4 weeks. In contrast to our CS1 and CS2 courses, these assignments have less scaffolding in place to help the students. While

they are given a large amount of detailed verbal and written advice about design and implementation during the course of the project, these students need to design their own solutions. The students then need to write accurate solution code along with their own test cases for locally verifying their solution code. An online auto-grader is then used to assess the correctness of students' solution code. The auto-grader also evaluates the code coverage of the student-written test cases. In case of a faulty solution code, students need to identify bugs and fix them using one or more debugging techniques of their choice. In many ways, their experience level and behavior are more like that of professional programmers than that of CS1/CS2 students.

We hypothesize that these students use different techniques for their testing and debugging. This depends on an individual student's prior knowledge, ability, and/or personal preferences. The course requires students to write their own JUnit test cases, but it is possible to use other techniques to test one's solution code. Nearly all of these students use the Eclipse IDE. Eclipse provides an environment for writing solution code along with JUnit tests, and a source-level debugger. We knew from prior experience that many students use console print statements to check code flow and variable values as a form of testing and debugging. We also believe that there can be additional techniques that students use for testing and debugging their code.

Our research goal is to understand the different bug investigation techniques used by intermediate-level students, and measure the effectiveness of such techniques in large programming assignments. We define *bug investigation* to be the holistic process of identifying the existence of one or more bugs in the solution code and fixing them. The process can consist of multiple iterations of testing and debugging.

We used a mixed-methods approach to get a better understanding of how students carry out their bug investigation. First, we conducted semi-structured interviews with 12 students from the course for qualitative analysis, with these students selected to represent a range of previous performance. Based on the interviews, we learned about different techniques that the students use for bug investigation. Second, we conducted an online survey on two semesters for the same course, to observe if our findings from the interviews reflected wider population. Finally, we collected empirical data about the different techniques that these students used to complete their projects. We hypothesize that students use testing and debugging closely as a part of a bigger process, bug investigation, to identify and fix bugs. We ran a thorough analysis on the collected data to find the extent to which our hypotheses were supported.

In our paper, we address the following research questions.

(1) What are the different techniques that the students use for bug investigation?
(2) How is the use of each bug investigation technique distributed among the student population?
(3) What is the relationship between different bug investigation techniques and project outcomes, like project correctness, and time taken?

By analyzing both qualitative and quantitative data, we aim to better characterize the bug investigation process practiced by these students.

**Structure of the paper.** In Section 2, we present related work on debugging techniques and strategies. We present our methodology in Section 3 and the corresponding analysis in Section 4. We discuss our results in Section 5 and consider possible threats to validity in Section 6.

## 2 RELATED WORK

Prior research has attempted to understand the various bug investigation techniques used by students [1, 5, 9, 11, 16, 22]. The literature ranges from studies conducted on small assignments done by small numbers of students to studies of large project assignments done by hundreds of students. The majority of these works focus on novice student programmers who are in an early stage of learning to write code, write test cases, and perform debugging in small assignments like those found in CS1/CS2 courses. In contrast, our work focuses on students who have an intermediate level of programming knowledge and expertise, and their process of finding and solving bugs in larger, more complex programming assignments. This requires a more complex analysis of the different testing and debugging practices exhibited by the students.

Murphy, et al. [16] conducted a qualitative analysis of different debugging techniques used by novice-level Java programmers. They categorized these techniques into a list of strategies. They then distinguished the strategies as good, bad, or quirky based on their effectiveness, productivity, and student's performance. Murphy, et al. also presented teaching guidelines that can help novice students learn and apply different debugging techniques effectively.

Gugerty and Olson [11] report two experiments to distinguish the debugging practices and strategies used by expert and novice programmers. In their first experiment, they compared sixteen first or second level students enrolled in an introductory Pascal course (novices) against eight advanced CS graduate students (experts). In their second experiment, they compared ten introductory undergraduate students against ten CS graduate students. Surprisingly, they found that both the novices and the experts used the same debugging techniques and strategies. They found that the experts showed superior performance by better understanding the code base before making any edits. The experts were also much faster in code comprehension compared to the novices. Finally, the experts were careful not to add additional bugs while looking for an existing bug. However, novices often accidentally introduced new bugs during their debugging process.

Ahmadzadeh, et al. [1] studied debugging patterns in students taking an introductory Java programming course. The authors considered students from two groups, 1) good programmers - who obtained more than 70% overall marks in the semester, and 2) weak programmers - who obtained less than 40% overall marks. The authors then designed an experiment where the students were asked to find and fix certain bugs (compiler and logical errors) in an author-written program within a fixed amount of time (120 minutes). Based on their performance in the experiment, the authors further categorized these students into 1) good debuggers, and 2) weak debuggers. The authors found that most of the good debuggers were good programmers. However, the opposite is not true, as less than half of the good programmers were found to be good debuggers. The authors posited that good debuggers have a separate

skill set from good programmers. They have a sound understanding of the code implementation and the bug. The authors measured this skill set by two factors: 1) the relevance of the changes made during the debugging process, and 2) their ability to isolate the bug. Thus, Ahmadzadeh, et al. found that otherwise good programmers, who lacked an understanding of the implementation of someone else's code, were unable to debug the code successfully. Students in a study by Katz, et al. [13] used different techniques for debugging and their choice of techniques varied based on whether they were debugging on their own code or other students' code.

There has been research to understand the debugging mindset and strategies among programmers [3, 5, 10, 17, 18, 20, 22, 23]. Nagy, et al. [17] found that student programmers follow a repetitive cycle upon finding a bug in their code. They follow the cycle of making small changes, resubmitting the code, and hoping the code works properly. Araki, et al. [3] suggested a set of initial hypotheses from which the programmers select one hypothesis at a time until they can verify the correct hypothesis behind a particular bug. More recently, Zeller, et al. [23] proposed a seven step approach for systematic debugging called TRAFFIC. The approach contains all the steps of debugging starting from the discovery of a bug to the ultimate fix for the bug. Chmiel and Loui [5] developed a model of debugging abilities and habits. This is based on the five stages of the Dreyfus model of skill acquisition [6], such as novice, advanced beginner, competent, proficient, and expert. This study [5] was conducted on students taking a sophomore-level assembly programming course, which means a typical student would have previously taken one or two programming courses. The authors found that novice debuggers frequently generate the same type of bugs by repeating the same types of mistakes. Such students spend too much time debugging using their preferred debugging technique. In contrast, competent debuggers feel comfortable using a variety of debugging techniques. Although there are several techniques for debugging, there is not a single, perfect set of debugging techniques for everyone [22]. Chmiel and Loui [5] found that learning multiple debugging techniques could enhance a student's overall debugging skill. In our research, we focus on intermediate student programmers who are more likely to use multiple techniques.

Recent work [4, 19] investigates why programmers use different debugging tools. In both studies, researchers found the IDE's source-level debugger and diagnostic print statements to be the most common debugging tools among professional programmers. Beller, et al. [4] found that 81.3%, 72.2%, and 71.6% of 176 professional programmers use a source level debugger, log files, and print statements, respectively. Perscheid, et al. [19] conducted an online survey of both professional and student programmers. According to the survey, student programmers, unlike professional programmers, tend to use print statements more than source level debuggers. In this research paper, we explore the extent to which the student programmers use different testing and debugging techniques.

# 3 METHODOLOGY

We collected data on student bug investigation practices using a mixed-methods approach. First, we conducted semi-structured interviews based on our prior knowledge and intuition of how students practice testing and debugging during their course assignments. Then, we used findings from the interviews to drive larger-scale quantitative analysis of usage logs from students' Eclipse IDEs and their submissions to the Web-CAT auto-grader tool. In this section we describe the data collection tasks. We present our findings in Section 4.

## 3.1 Interviews with Students

*3.1.1 Study Subjects.* We conducted a total of 12 interviews. The interviewees were from three different sections of the CS3-level Data Structures and Algorithms course in Fall 2019, taught by two instructors. The three sections were taught in the same semester and the course materials, schedules, and evaluation criteria were identical for each section. The interviews were conducted after the students had completed their first project of the course. Due to scheduling, two of the twelve interviews were conducted after the completion of the second project. The students had a range of performance on the first project. Seven of the students received a score of 100 on their first project. The remaining five students scored less than 90 in their first project. The median score on this project was 92.7. Students were free to work with a partner or not, as they chose. Six of the interviewees completed the project by themselves, while the other six students did it with a partner.

*3.1.2 Study Design.* The interviews consisted of 60 minute sessions with 20 questions. Some of the topics in the questionnaire are as follows.

(1) How difficult was the project? What were the new concepts learned?
(2) What bug investigation techniques were used for testing and debugging? What was their primary technique for testing and debugging?
(3) When was the first time they used bug investigation techniques?
(4) Did they use multiple bug investigation techniques?
(5) What were the drivers for choosing between multiple bug investigation techniques?
(6) How did they use the bug investigation techniques?
(7) Reflection on different bug investigation techniques.
(8) What were the difficulties that they faced regarding bug investigation?

The questions were used as a starting point for discussing different topics about the interviewee's bug investigation practices. The interviewees were asked to share examples as appropriate to answer each of the questions. They were also encouraged to add information, reflection, or experiences that they felt were relevant.

We conducted the interviews with the help of two authors of this paper. One of the authors transcribed the answers as another author guided the discussion with the interviewee. After the end of interviews, each interviewee was compensated with a $15 Amazon gift card for their time. We used Grounded Theory [21] to analyze the interviews. We manually coded the contents of the twelve interview transcripts. Then we clustered similar codes into concept groups, and further clustered the concept groups into categories.

According to our findings, students mainly used the following five bug investigation techniques.

**Table 1: The number of students using a single or a combination of bug investigation techniques.**

| Different Techniques | As Primary Tool | At Least Once |
|---|---|---|
| DPS | 7 | 12 |
| JUnit Testing | 2 | 5 |
| Debugger | 2 | 4 |
| Web-CAT | 1 | 4 |
| Manual Tracing | 0 | 3 |
| DPS + Debugger | 1 | 4 |
| DPS + JUnit + Debugger | 1 | 4 |

(1) Diagnostic Print Statements (DPS)
(2) JUnit Testing
(3) IDE Debugger
(4) Web-CAT Submission
(5) Manual Tracing

Table 1 shows the number of interviewees who mentioned using either one or a combination of multiple bug investigation techniques. The most popular technique was Diagnostic Print Statements (DPS). A DPS is a temporary printed output that a student uses for testing and/or debugging. According to our interviews, all twelve interviewees used DPS to some extent. Seven students stated that DPS was their primary tool for bug investigation.

The second most popular tool of choice was JUnit testing. Students were required to write JUnit tests since they were graded in part on how well their JUnit test suite performed on code coverage metrics. Five of the interviewees mentioned that they used JUnit testing to some extent for testing their solution code and to localize a bug. Among these five students, two of them used JUnit tests as their primary bug investigation technique. This technique was considered more useful when working with a partner. Two of the students mentioned that they wrote JUnit tests to verify the correctness of code that was written by their project partner.

We noticed that the four interviewees, who used more than one bug investigation techniques, were among the high scoring students. Chmiel and Loui [5] found that competent debuggers use multiple debugging techniques effectively by alternating them. The four interviewees who mentioned using source-level debugger all had prior experience with the debugger from their earlier CS courses, unlike the rest of the interviewees. In agreement to Chmiel and Loui [5], we found that students using multiple techniques have additional skill set.

Web-CAT is not generally considered to be a testing or debugging tool. It is an automatic grading system that instructors use to evaluate student performance on programming projects, by calculating the correctness of their solutions, evaluating student-written test cases using code coverage, and assessing documentation and other aspects of programming style. There was no limit to the number of times a student could submit to Web-CAT. Some students attempted to debug their solution code by submitting to Web-CAT, making changes, and then resubmitting multiple times until Web-CAT verified that the grader test cases passed. Four of our interviewees mentioned using Web-CAT to verify their solution code. Three of them mostly used it for cases where the output format of their

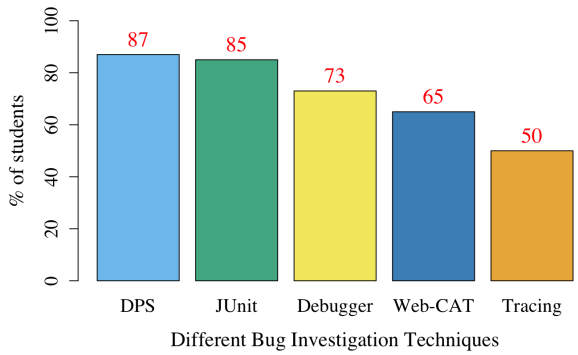


**Figure 1: From our online survey: percentage of students using different bug investigation techniques.**

program did not meet the assignment requirements. Only one interviewee mentioned that they used Web-CAT as a primary bug investigation technique.

Students also reported that they found bugs by manually tracing through their code. Three of our interviewees mentioned they used manual tracing, but not as their primary tool. According to one interviewee:

"[Manual Tracing]...with my partner helped me figure out the logical flaw in our code."

Some students reported using a combination of techniques. According to these interviewees, they used different techniques for different purposes. For example, JUnit testing might be used to determine the existence of a bug, DPS for finding the bug's location, and the IDE's source-level debugger for uncovering faulty program logic.

During our interviews, all of the interviewees mentioned that they faced some level of difficulties with bug investigation. The most common difficulty faced (7 out of 12) was localizing the bug. Five of the interviewees believed that the difficulties they faced could be solved by improving their debugging strategies and practices. For example:

"Probably more practice [could help]."

"Try something else instead of trying the same thing over and over again and expecting different results."

"More time, more incremental development, more test cases, probably using debugger could help."

Therefore, based on Xie and Yang [22], we believe that a good portion of the student programmers suffer from ineffective practices for bug investigation, and that they can be improved upon.

## 3.2 Online Survey

Analysis of the interviews lead us to the following hypotheses.

(1) H1. The majority of the students use DPS.
(2) H2. Students use a combination of multiple techniques for bug investigation.
(3) H3. Students who use multiple techniques tend to perform better.
(4) H4. Some students use ineffective practices for bug investigation.

We wanted to see if our findings from the interviews reflected the behavior of a wider population. In order to test this, we designed an online survey.

*3.2.1 Study Subjects.* We tailored our survey questions towards students in the same CS3-level Data Structures and Algorithms course. We conducted our online survey during two course offerings, Spring 2020 and Summer 2020. The spring semester was 16 weeks long, whereas, the summer semester was 6 weeks long. For Spring 2020, there were two sections, taught by two different instructors. The course was conducted in person for half of the semester, then moved online for the rest of the semester due to the COVID-19 pandemic. There were a total of 247 students in two sections for the Spring 2020 semester. The course materials, project specification, and exams were the same for both sections. For Summer 2020, the course was offered fully online from the start to a total of 41 students. This course was a combined course consisting of both undergraduate (38) and graduate (3) students. The course was taught by a different instructor from Spring 2020 semester. Having said that, the course contents and the projects were similar to the ones taught in Spring 2020 and Fall 2019.

We received a total of 47 responses from Spring 2020 semester and a total of 20 responses from the Summer 2020 semester. Participation in the survey was voluntary and there was no compensation for participation. The survey responses were recorded anonymously. Responses were recorded from unique IP addresses to make sure one participator filled up the survey only once.

*3.2.2 Study Design.* The survey was designed to be answered using any browser or phone and took 5 to 10 minutes to complete. The average time for the survey participation was approximately 6 minutes. We asked various questions regarding bug investigation, such as what debugging techniques were used, which techniques were found to be the most useful, and which techniques were the easiest to use.

Our survey results were consistent with the findings from our interviews. Based on our 67 responses, we found that the majority of participants, 87%, used DPS. This supports our first hypothesis, H1, that the majority of the students use DPS for bug investigation. The other techniques — such as JUnit testing, IDE's source-level debugger, Web-CAT submission, and manual tracing — were used by 85%, 73%, 65%, and 50%, respectively. 96% of the participants indicated that they used a combination of DPS, JUnit test cases, IDE debugger, and/or the Web-CAT auto-grader. This supports our second hypothesis, H2, that the students use a combination of multiple techniques for bug investigation. Furthermore, we found that 81%, 73%, and 50% of the participants found DPS, JUnit testing, and IDE debugger, respectively, to be useful debugging tools. Only 12% of the participants found Web-CAT to be useful as a bug investigation tool.

Since writing JUnit test cases were required by the course, we wanted to find how many of the students believe they would use JUnit testing if it were not required. 50% of the participants agreed that they would write JUnit test cases even if it were not required. 31% of the participants disagreed, and 19% of the participants neither agreed nor disagreed. We also found that 84% of the participants used JUnit testing for actual testing and/or debugging purposes (not just for the course requirement). Finally, we found that a majority of the participants, 62%, found finding the bug to be the most difficult part of bug investigation, compared to fixing the bug (50% participants).

## 3.3 Analysis of Development Process Data

*3.3.1 Web-CAT Online Auto-grader Data.* For this research, we collected data from three sections of the CS3-level Data Structures and Algorithms course. We analyzed all of the submissions made towards Web-CAT. Each submission was graded automatically against the instructors' hidden reference tests. We also automatically graded the quality of student test cases as measured by their code coverage, and automatically graded commenting and related style issues. TAs manually graded projects on adherence to the design requirements. The overall project score is calculated by weighted summation of these metrics. For our research, we are most interested in the project correctness score which is the correctness of student code measured by instructors' reference tests. This can be viewed as a summative assessment of the whole design, implementation, testing, and debugging process.

*3.3.2 Event-level data via the DevEventTracker Plugin.* To analyze students' incremental development process and their detailed bug investigation behavior, we collected students' development data using DevEventTracker [14]. This Eclipse plugin collects students' development events from their local machine. Students are recommended to use the plugin for this course and are asked for their consent for using their development data anonymously. We only included the data from the students who provided consent. We collected over 6 million development events via DevEventTracker. DevEventTracker records timestamped event data, such as adding, removing, and editing of solution code and test code, save events, etc. It tracks launch event data, such as normal launch events, debugger launch events, and JUnit test launch events. Specific debugger event data is tracked, such as setting breakpoints, step into, step over, etc.

*3.3.3 Code Snapshot Repository Data.* To analyze bug investigation techniques inside solution code, we used the DevEventTracker plugin to collect code snapshots of the solution code and the test cases throughout the students' development process. These code snapshots contain all the temporary lines of code that students write during their project development. From this collection of code snapshots, we were able to identify DPS. We developed a rule-based classifier to distinguish DPS from print statements required by the assignment (non-DPS). Our classifier identifies a statement as a DPS using the following rules:

(1) A DPS cannot be an empty statement, for example, *'System.out.println()'*, or whitespace, for example, *'System.out.print("\n")'*.
(2) A DPS cannot appear in the final submission.
(3) It is not a DPS if it is required by the assignment specification.

If the criteria above are met, then we can classify the statement as a potential DPS.

We devised a program based on the open-source Java package RepoDriller [2] that drilled through the code snapshots of all the projects. We then used the rules above to identify the DPS in a

given student project. We noted both the insertion and removal of a DPS. The removal of a DPS can be done by commenting the print statement or by deleting it. Then, we counted the number of DPS for each project to calculate the extent to which a student uses DPS.

To check the effectiveness of our rule-based DPS classifier, we conducted a preliminary experiment. We randomly selected three submission sets for each of the four project assignments over the entire semester. Therefore, we had a set of 12 project submission sets. We gathered all the print statements from our 12 sample submission sets as described above, collecting a total of 1,467 print statements from the 12 project histories (mean = 122 and s.d = 89). One of the authors of this paper then manually classified DPS from the total 1,467 print statements. We ran our rule-based DPS classifier separately on the same print statement collection, and then compared the two classifications. Our automated classifier correctly identified all 611 DPS, indicating that it could be reliably used on the entire corpus of project histories.

For our final step, we processed the three data collections to make them compatible. We had project scores for all the students from the Web-CAT system. However, using the DevEventTracker plugin was voluntary, and was used by 58% of students. Since our analyses require data that are common across all three collections, we only considered students who had used the DevEventTracker plugin.

We investigated the difference in project scores between the four projects. We considered the project scores of the students who used the DevEventTracker plugin. We ran a Kruskal–Wallis test on the project scores across all four projects. We found no significant difference between the project scores ($p-value = 0.085$). Therefore, we used data from all four projects.

## 4 ANALYSIS

In this section, we report on an analysis of the three data collections, aimed at understanding the bug investigation techniques used by intermediate student programmers. Our Web-CAT submission collection includes submissions from 222 students. All of these students submitted at least once for Project 1. However, only 184 students submitted at least once for Project 2, a drop by 17%. Project 3 was submitted at least once by 182 students, and finally, Project 4 was submitted by 175 students. This is a total drop over the course of the semester of 21% from Project 1. [1]

We notice that there were 38 students who dropped the course after the end of Project 1. 35 of these 38 students used neither the DevEventTracker plugin nor the Git snapshots. Therefore, we had no access to the incremental development data of the majority of the dropped out students. Upon inspection we found, as expected, that the project scores were significantly lower among the dropped students. The mean project correctness score among the dropped students was 58% — which is significantly lower than the mean of the rest of the class, 84%, with a $p < 0.0001$.

Figure 2 shows the overall performance on the programming projects of all the students over the semester. We can see that, while overall the scores are good, there are several students who struggle compared to the rest of the class.

[1]This suggests that a fair number of students struggle to stay in the course or perform satisfactorily.

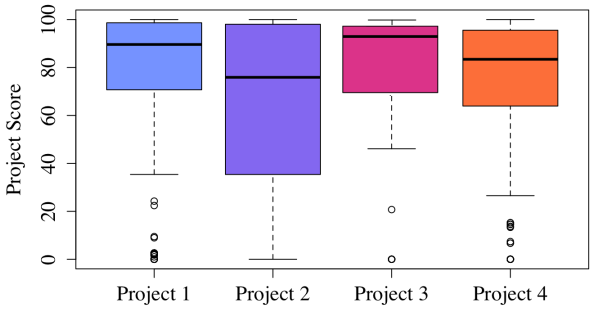


**Figure 2: Box plots of the score distribution for each project. The box shows the 75th and 25th percentiles.**

**Table 2: Different bug investigation techniques and the percentage of students using them across all the four projects. ⋆ = 1 user.**

| Different Techniques | Used by students | Mean Score | Median Score | Standard Deviation |
|---|---|---|---|---|
| No Technique | 2% | 72 | 95 | 44.92 |
| DPS (alone) | 4% | 65 | 79 | 36.38 |
| JUnit (alone) | 3% | 81 | 90 | 35.59 |
| Debugger (alone) | 0%⋆ | 93 | 93 | 0 |
| DPS +JUnit Testing | 18% | 72 | 84 | 32.68 |
| DPS + Debugger | 2% | 64 | 78 | 38 |
| JUnit + Debugger | 8% | 79 | 91 | 32.83 |
| DPS + JUnit + Debugger | 63% | 78 | 90 | 28.79 |

To understand students' testing and debugging practices, we analyzed our collected data to count the number of students using the different bug investigation techniques during their project development process. We focused on the techniques — DPS, JUnit testing, and IDE debugger — that were measurable in our data. From the code snapshot data, we found that 87% of the students used DPS to some extent in their projects, as shown in Table 2. We also found that 92% of the students used JUnit testing and 73% of the students used IDE debugger. This contradicts our hypothesis, H1, which was based on the interview and survey results. So, for our first hypothesis, H1, we found that the majority of the students use JUnit testing, closely followed by DPS.

We also found that there are students who use multiple techniques for bug investigation during their project development process. From Table 2, we can see that 91% of the students used more than one technique for bug investigation. Similarly, 63% of the students used a combination of all three techniques — i.e., DPS, JUnit testing, and IDE debugger — in their projects. This supports our second hypothesis, H2.

In support of our third hypothesis H3, we found that students who used multiple debugging techniques over the course of a

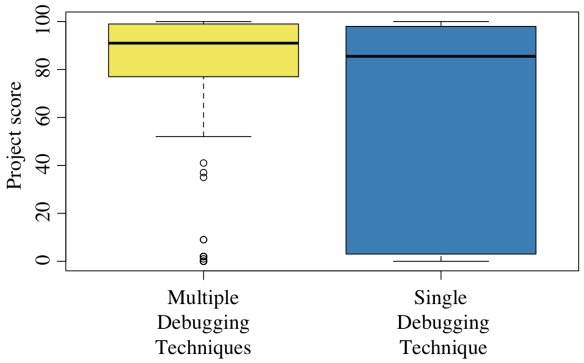


**Figure 3: Boxplots of project score for students using (1) Multiple Debugging Techniques (any combination of DPS, JUnit, and Debugger), and (2) Single Debugging Technique (any one of DPS/JUnit/Debugger). Students using a combination of multiple debugging techniques tend to have a higher project score then students using single debugging technique.**

project performed better (in terms of correctness) than students who only used a single debugging technique. Figure 3 depicts correctness score distributions for these two groups of students. On the left are scores from students who used multiple debugging techniques ($n = 201$), and on the right are scores from students who only used a single debugging technique ($n = 94$). We used Welch's two sample $t$-test to check for differences in project scores between the two groups. With $p < 0.001$, the test indicated that students who used multiple debugging techniques produced projects with higher correctness (mean = 84%, s.d = 28%) than students who only used a single debugging technique (mean = 64%, s.d = 41%). Note that it is possible for students to appear in both groups, e.g., if they used JUnit tests and DPS on Project 1 but then solely relied on JUnit tests on Project 2. So we found that students who used a combination of multiple bug investigation techniques tend to perform better in the projects. Therefore, our third hypothesis, H3, was supported.

We investigated if the project score is impacted by different bug investigation practices. We did not find any correlation between individual bug investigation techniques and project score. However, we found that students who use a combination of multiple techniques — such as both JUnit testing and the IDE debugger in their project — obtained higher mean correctness than the others. From Table 2, we can see that 8% of the total students (77 out of 429) use a combination of JUnit testing and IDE debugger as their bug investigation practice. This group of students obtained a median project score of 91% (s.d = 32.83). Furthermore, we can notice that students using JUnit alone or in a combination of other techniques tend to have median score of 90% or above. The exception to this pattern is the group of students who used both DPS and JUnit testing (median score 84). We can further observe that students using DPS alone or in combination tend to have lower median scores. The other two groups — no technique and IDE debugger alone — that achieve a higher median score than 90% are very small in number of students. Only 10 out of 429 students (2%) and only 1 out of 429 students (almost 0%) used no technique and IDE debugger respectively. Therefore, we consider them as outliers.

**Table 3: Three factor crossed ANOVA for examining the effect of different bug investigation techniques on the project score correctness.**

| | Df | Sum Sq | Mean Sq | F-value | *p-value* |
|---|---|---|---|---|---|
| DPS | 1 | 388 | 388 | 0.407 | 0.524 |
| JUnit | 1 | 3402 | 3402 | **3.562** | **0.05** |
| Debug | 1 | 1155 | 1155 | 1.210 | 0.257 |
| DPS:JUnit | 1 | 391 | 391 | 0.410 | 0.800 |
| DPS:Debug | 1 | 176 | 176 | 0.185 | 0.964 |
| JUnit:Debug | 1 | 37 | 37 | 0.038 | 0.844 |
| DPS:JUnit:Debug | 1 | 577 | 577 | 0.604 | 0.437 |
| Residuals | 421 | 402076 | 955 | N/A | N/A |

To further understand the correlation between project score correctness and different combinations of techniques, we ran a three factor crossed ANOVA. The result can be seen at Table 3. According to Table 3, we found JUnit testing alone to be the most important factor to project score correctness. This is statistically significant ($p - value = 0.05$). No other factor or interaction in Table 3 significantly correlates to project score correctness.

Next, we investigated the relationship between bug investigation techniques and the total time spent on the project. Following Kazerouni, et al. [15], we used DevEventTracker data to calculate the total time spent interacting with the Eclipse IDE (as a proxy for the total time spent "working on the project"). In agreement with Fenwick, et al. [8], Edwards, et al. [7], and Kazerouni, et al. [15], we found no correlation between project score and the number of hours spent on the project. A three-factor analysis of variance indicated that DPS usage (alone) and JUnit testing (alone) were significantly positively related to total time spent on the project (Table 4).

In order to understand practices of bug investigation outside the IDE event-level data, we considered the number of times a student submits to Web-CAT. We found that students who used more than one bug investigation technique tended to complete their project with fewer submissions to Web-CAT. We ran a three-factor cross ANOVA on the effect of different techniques on the number of Web-CAT submissions. We found sufficient evidence to suggest that using only a single technique — DPS, JUnit, or IDE debugger — correlates to a higher number of submissions to Web-CAT. This suggests that the students practicing only a single technique tend to rely on other means, such as submitting to Web-CAT, for testing and debugging their solution.

A large number of Web-CAT submissions in a short period of time could indicate that the student is using Web-CAT's ability to serve as an "oracle" as a bug investigation tool. In this case, a student might use Web-CAT for testing purposes, i.e., to identify if there is a bug in the code or if the code misses any edge cases. For this reason, a student would make multiple unsuccessful submissions to Web-CAT consecutively. In contrast, a low number of Web-CAT submissions in a long period of time (assuming that they eventually received a good correctness score) could indicate that a student used other techniques for bug investigation. From Figure 4, we

**Table 4: Three factor crossed ANOVA for examining the effect of different bug investigation techniques on the hours spent on each project.**

| | Df | Sum Sq | Mean Sq | F-value | *p-value* |
|---|---|---|---|---|---|
| DPS | 1 | 18853 | 18853 | **12.430** | **< 0.001** |
| JUnit | 1 | 20992 | 20992 | **13.841** | **< 0.001** |
| Debug | 1 | 1955 | 1955 | 1.289 | 0.257 |
| DPS:JUnit | 1 | 97 | 97 | 0.064 | 0.800 |
| DPS:Debug | 1 | 3 | 3 | 0.002 | 0.964 |
| JUnit:Debug | 1 | 377 | 377 | 0.249 | 0.618 |
| Residuals | 316 | 479260 | 1517 | N/A | N/A |

can see that there is a positive correlation between the number of Web-CAT submissions made and total time required to complete the project (where $p < 0.001$; $R^2 = 17\%$).

To further extend this idea, we categorized the overall project development process into two phases, 1) new code generation, and 2) bug investigation. New code generation occurs when a student writes additional code to implement new functionality. Bug investigation is when a student tries to find and fix one or more bugs in the existing code with the help of different testing and debugging techniques. The students are recommended to write their solution code, test the new code by writing their own test cases, and then submit to the Web-CAT auto-grader. However, some students write new code and then submit directly to Web-CAT, bypassing the step of testing with their own test cases. In such cases, students use the Web-CAT auto-grader as a way to test their solution code, by adopting a trial-and-error approach. This generally gives rise to multiple submissions to the Web-CAT system with often little to no improvement to the project correctness. We identify this behavior as "stagnant Web-CAT submission" when multiple consecutive Web-CAT submissions fail to improve the overall project score.

In our analysis, we found that students use multiple techniques in the bug investigation phase. We found the most common technique to be launching the code with the presence of one or more diagnostic print statements. The other common techniques are launching the IDE's source-level debugger and running the JUnit tests. We also found some cases where the student submitted to Web-CAT without using any of the above bug investigation techniques. We identify these cases as using the Web-CAT auto-grader as a testing and/or debugging tool. We found that submitting to Web-CAT without using additional bug investigation techniques (DPS, IDE debugger, and/or JUnit testing) tends to result in more "stagnant Web-CAT submissions" (where $p < 0.001$; $R^2 = 68.25\%$). We also found that projects with a higher number of "stagnant Web-CAT submissions" took longer to complete (where $p < 0.001$; $R^2 = 30.72\%$). As a result, we can conclude that relying on Web-CAT as a form of testing and debugging could lead to more time spent on the overall project. However, we found no evidence to correlate the project score with the number of "stagnant Web-CAT submissions".

We compared the influence of practicing single technique vs multiple techniques by the same student. We found that students used more techniques in the earlier two projects and settled in using one or two techniques in the later two projects. For example, 73 and 85 students used a combination of all three techniques (DPS + JUnit + IDE debugger) in project 1 and 2 respectively. For project 3 and 4, this number got reduced to 56 students. There were 12 students who used all three techniques across all four projects. We found no significant change in project score or project activity time as students used fewer techniques in their development process.

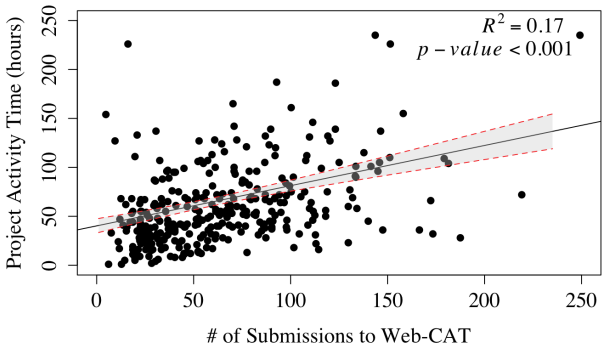


**Figure 4: Project activity time (in hours) vs the number of total Web-CAT submissions for all four projects. There is a significant positive relationship, where $p < 0.001$; $R^2 = 0.17$.**

Finally, to address our fourth hypothesis, H4, we can suggest that submitting to Web-CAT without first doing local testing and debugging is an ineffective practice for bug investigation.

## 5 DISCUSSION

Bug investigation, i.e., testing and debugging, is an integral part of project development. This is especially true for a large and complex project, such as the ones in the CS3-level course that we studied. Furthermore, there are several effective ways to investigate bugs. Based on the current literature, currently there is no underlying theory that explores the bug investigation nature in intermediate student programmers. Most of the current research regarding debugging is focused on novice or professional programmers [1, 4, 5, 9, 16, 19]. In our research, we tried to explore different techniques that the intermediate student programmers use for bug investigation. Based on our qualitative study, we formulated four hypotheses. We then used the students' quantitative usage data to verify our initial hypotheses.

*H1. The majority of the students use DPS.*

Using DPS to test and debug is one of the simplest forms of bug investigation. In addition to its simplicity, it is one of the fastest and easiest techniques. In our interview studies, we found that all of the students used DPS to some extent. Many attributed its fast nature of adding/deleting print statements as the prime reason for using it. Reviewing students' code snapshots, we found that over 87% of the students used DPS while developing their projects. We found that even more students (92%) used JUnit testing. This is because writing JUnit test cases were required in the project. This, inherently, gave rise to more JUnit launches as students tested their test cases. According to our online survey, only 84% of the students use JUnit testing for testing and/or debugging purposes. The remaining 16% of the students use it as a course requirement. 31% of the survey respondents also mentioned they would not use

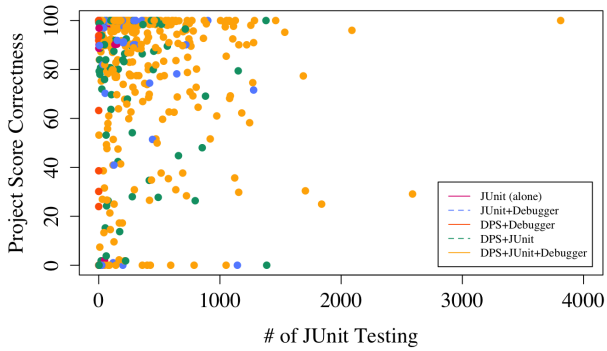


**Figure 5: Plot of number of total test launches vs project score.**

JUnit testing if it were not a course requirement. Therefore, we can say that the majority of the students used DPS for bug investigation purpose. This is congruent with the study conducted by Perscheid, et al. [19].

*H2. Students use a combination of multiple techniques for bug investigation.*

The projects for this course are relatively large and complex in nature. Students are required to use new knowledge as well their past experiences in programming. Additionally, since the students are intermediate-level programmers, they tend to be advanced beginners to expert in the Dreyfus model of skill acquisition [6]. Therefore, we can assume that intermediate students would utilize multiple techniques to find and fix bugs in their code. We first formulated this hypothesis from our interviews. We found 8 of 12 student interviewees used more than one technique. Two students listed a combination of different techniques as their primary tool of bug investigation. To further verify this hypothesis, we used students' usage data and found over 91% students used a combination of DPS, JUnit testing, and IDE debugger. This proves that many intermediate student programmers use a combination of multiple techniques for bug investigation. This supports the findings of Katz, et al. [13] and Chmiel and Loui [5], where they found that student programmers use multiple and different debugging techniques.

*H3. Students who use multiple techniques tend to perform better.*

Beyond discovering what intermediate-level student programmers *do*, we also want to know if their bug investigation practices have any influence in the overall project performance. We formulated our third hypothesis to check if using multiple techniques has an effect on project performance. We considered both project score and project activity time as the metrics of performance. Since each bug investigation technique requires additional skills, we can suggest that students who used multiple techniques are more skilled than students who used only a single technique. The data support our third hypothesis, that the students who use multiple techniques tend to perform better in project score and take less amount of hours to complete the projects. This supports and extends the findings of Chmiel and Loui [5] conducted on novice programmers.

We analyzed the project perfomance — score correctness and activity time — against all the combinations of different bug investigation techniques. We found indications that the usage of JUnit testing influence project performance. From Table 3, we can see that using JUnit is the most important factor when it comes to project score. We analyzed the effect of using JUnit test cases (alone or in combination with other techniques) on project performance as measured by project score. [2] From Figure 5, we can observe the relation between project score and JUnit usage. The data points in Figure 5 are color-coded to represent the different combinations of bug investigation techniques. We found no significant correlation between project score and different combination of JUnit usage.

We also found that the students, who use multiple techniques, are also efficient in using local techniques — such as DPS, JUnit, and IDE debugger — for localizing and fixing bugs, and verifying their solution. The low number of submissions made to Web-CAT supports this notion. On the contrary, students using only single technique for bug investigation tend to rely on the auto-grader for verifying their solution. From Figure 4, we can see that relying on Web-CAT takes more time to complete the project. Hence, we can suggest students, who use multiple techniques, tend to perform better by effectively using the local techniques. This further supports the results of Murphy, et al. [16] and Chmiel and Loui [5].

*H4. Some students use ineffective practices for bug investigation.*

We formulated this hypothesis based on the literature and our interviews. We learned that students consider debugging to be one of the hardest parts of programming. Students mentioned several cases where they wished that they had done the bug investigation differently. We investigated different practices among the students, namely frequent Web-CAT submissions, frequent use of JUnit test cases, and frequent use of the IDE's source-level debugger. We further investigated "stagnant Web-CAT submissions" and found with significant evidence that it leads to longer time to complete the projects. We suggest that the practice of frequently using Web-CAT tends to make project progress slower. Therefore, extending the bad debugging practices found by Murphy, et al. [16], we can conclude that there exist students with ineffective practices for bug investigation, namely, over-reliance on the auto-grader as an "oracle" for correctness.

## 6 THREATS TO VALIDITY

We now consider some possible threats to the validity of our work. One threat to our qualitative analysis is that we might not have included a representative sample of student behaviors. For our qualitative study, we interviewed 12 intermediate-level student programmers. We randomly selected 12 students from two sections of the CS3 Data Structures and Algorithms course offered in the same semester, Fall 2019. We found that the students were reasonably distributed with respect to performance on the first project. Seven of the twelve students scored 100 or higher in the first project. A student can get 100 points for correctness, style, and design. The class median score for the first project was 92.7. The remaining five students struggled with their first project and scored below

[2]Here, the total JUnit usage is calculated by the number of times a student launched a JUnit session in their Eclipse IDE.

90. Therefore, we believe that we covered both high performing students and low performing/struggling students in our interviews.

Another possibility is that our attention might be focused on the wrong questions when we did our quantitative analysis. In particular, we had choices on how we divided students into comparison groups. We began with the research literature to help us formulate our underlying theory. However, we could not find any underlying theory regarding bug investigation techniques in intermediate student programmers. Therefore, we based our initial understanding from the research conducted on novice students. Based on this, we prepared interview questions. We then conducted our semi-structured interviews to formulate our hypotheses via qualitative research methods. Finally, we verified our hypotheses quantitatively using students' event-level data, code snapshots, and submission data. Therefore, we used mixed-research method to solidify our results.

We conducted our online survey across two semesters, Spring and Summer 2020. The Summer semester was shorter (6 weeks) compared to the Spring semester (16 weeks). The Summer semester required three projects, whereas the Spring semester required four. The projects offered during each session had similar specifications and difficulty. We conducted the online survey after the end of Project 1 to avoid additional biases. Responses from each semester resulted in similar findings.

We developed our understanding of "stagnant Web-CAT submissions" based on literature review, prior preliminary research, and student interviews. However, students may have various plausible causes for these stagnant submissions. For example, students can submit their incomplete solution code to Web-CAT for checking how much partial credit they might get. These submissions are more common shortly before the deadline and are not necessarily for bug investigation purposes. Therefore, we focused on the stagnant Web-CAT submissions which are not right before the deadline. This could be further researched by exploring the actual changes made by the students in these submissions. We plan to explore this in our future research.

## 7 CONCLUSION AND FUTURE WORK

In this paper, we explored different bug investigation techniques in intermediate-level student programmers using mixed-research methods. We used interviews to form our foundation in understanding the techniques used by students for bug investigation in a post-CS2 Data Structures and Algorithms course. We then used students' event-level usage data to verify our understanding quantitatively. We learned that students collectively used different techniques for investigating bugs, namely diagnostic print statements, unit testing, a source-level debugger, submissions to an auto-grader, and manual tracing. We then found that the simplest technique, DPS, is one of the most commonly used technique. We further found that some of these students routinely use a combination of different bug investigation techniques. Students using such a combination tend to have better project performance. Finally, we tried to discover some ineffective bug investigation practices, and found that relying on the auto-grader as an "oracle" for correctness is an inefficient way to debug.

For our future work, we plan to explore more on the use of multiple debugging techniques. We plan to develop a metric to quantify the level of use for different techniques. We also plan to identify more ineffective bug investigation practices. This would allow us to provide feedback to students to discourage the bad practices. Finally, we hope to provide positive feedback to encourage students to keep on improving their testing and debugging skills.

## REFERENCES


[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. 84–88.

[2] M Aniche. 2012. RepoDriller.

[3] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. 1991. A general framework for debugging. *IEEE software* 8, 3 (1991), 14–20.

[4] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior Among Programmers. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 572–583. https://doi.org/10.1145/3180155.3180175

[5] Ryan Chmiel and Michael C Loui. 2004. Debugging: from novice to expert. *ACM SIGCSE Bulletin* 36, 1 (2004), 17–21.

[6] Hubert L Dreyfus and Stuart E Dreyfus. 1986. The power of human intuition and expertise in the era of the computer. *Mind over machine. Nueva York: The Free Press* (1986).

[7] Stephen H Edwards, Jason Snyder, Manuel A Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. 2009. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the fifth international workshop on Computing education research workshop*. 3–14.

[8] James B Fenwick Jr, Cindy Norris, Frank E Barry, Josh Rountree, Cole J Spicer, and Scott D Cheek. 2009. Another look at the behaviors of novice programmers. *ACM SIGCSE Bulletin* 41, 1 (2009), 296–300.

[9] Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116.

[10] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2009. Debugging from the student perspective. *IEEE Transactions on Education* 53, 3 (2009), 390–396.

[11] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 171–174.

[12] Brent Hailpern and Padmanabhan Santhanam. 2002. Software debugging, testing, and verification. *IBM Systems Journal* 41, 1 (2002), 4–12.

[13] Irvin R Katz and John R Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.

[14] Ayaan M Kazerouni, Stephen H Edwards, T Simin Hall, and Clifford A Shaffer. 2017. DevEventTracker: Tracking development events to assess incremental development and procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 104–109.

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

[16] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin* 40, 1 (2008), 163–167.

[17] G Nagy and MC Pennebaker. 1971. A step toward automatic analysis of logically undetectable programming errors. In *Technical Report RC 3407*. IBM Thomas J. Watson Research Center Yorktown Heights, NY.

[18] Devon H O'Dell. 2017. The debugging mindset. *Queue* 15, 1 (2017), 71–90.

[19] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2017), 83–110.

[20] Michael J Scott and Gheorghita Ghinea. 2013. On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice. *IEEE Transactions on Education* 57, 3 (2013), 169–174.

[21] Anselm Strauss and Juliet Corbin. 1994. Grounded theory methodology. *Handbook of qualitative research* 17 (1994), 273–85.

[22] Min Xie and Bo Yang. 2003. A study of the effect of imperfect debugging on software development cost. *IEEE Transactions on Software Engineering* 29, 5 (2003), 471–473.

[23] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.