Exploring the Barriers and Factors that Influence Debugger Usage for Students

Minhyuk Ko, Dibyendu Brinto Bose, Hemayet Ahmed Chowdhury, Mohammed Seyam, Chris Brown
Department of Computer Science
Virginia Tech, Blacksburg, VA, USA
{minhyukko, brintodibyendu, hemayetahmedc, seyam, dcbrown}@vt.edu

***Abstract*—Debugging is one of the most expensive and time-consuming processes in software development. To support programmers, researchers, and developers have introduced a wide variety of debuggers or tools to automatically find errors in code, to make this process more efficient. However, there is a gap between industry developers and students regarding the skillful use of debuggers. We aim to understand this gap by studying barriers that hinder new programmers from using debuggers. We conducted a survey involving 73 students with various extents of programming experience and performed qualitative analysis. The goal was to extract insights into why students do not develop debugger usage skills. Our results suggest the general lack of academic course focus on debuggers is one of the primary reasons for avoidance. At the same time, complex user interfaces and a lack of visualization also seem intimidating for many students, making using a debugger unappealing. Based on the results, we provide guidelines to motivate future debugger designs and education materials to improve debugger usage. Our survey results summary is publicly available at https://github.com/minhyukko/vlhcc23**

***Index Terms*—Debuggers, Novice Programmers, Barriers, Education**

I. Introduction

Debugging is critical in software development, however most programmers find the process of finding and fixing errors in source code to be difficult and demanding [20]. Software engineers often spend more time debugging software problems in the code than writing code [2], and accelerating this process can increase the productivity of programmers as well as the quality of the program [21]. Moreover, debugging is a critical part of learning how to program, yet it is one of the most challenging activities for Computer Science (CS) students to grasp [7].

To make debugging more efficient, developers and researchers have introduced a wide variety of *debuggers*, or automated tools with functionality to support various debugging tasks, such as locating bugs in code to repair the program. For example, Crisp is a tool for Java that allows developers to review parts of code for debugging based on failing test cases [4]. Debuggers have also been shown to help student programmers complete debugging tasks significantly faster [5].

However, despite the existence of automated debugging tools, software engineers often avoid using debuggers to find and fix errors in practice. There are many issues with automated debugging techniques that lead them to be ignored by developers, with prior work suggesting professional software engineers largely find automated debugging techniques unhelpful [13].

To that end, programmers adopt less efficient approaches such as *print statement debugging*, also known as "printf debugging" [2], or the process of manually inserting print statements in the code and observing the output to determine the location of errors. While this practice is frequently used by students and novices [12], experienced programmers also acknowledge using print statements for debugging their code [9].

This motivates us to study barriers to the use of debuggers. Specifically, we aim to understand what influences decisions to use tools or manual print statements. In that context, we conducted a survey and received responses from 73 software engineering students with varying degrees of programming experience. Our results suggest that effort and lack of education are the main barriers to debugger usage, despite success and positive experiences when using debuggers to resolve bugs. This work contributes a detailed analysis of students' debugging behaviors and decisions, and our findings provide implications for improving debugging tool development and programming education to increase knowledge and usage of debuggers for automating debugging tasks.

II. Related Work

Previous studies have been conducted to explore debugging practices by novice programmers. Katz and Anderson [10] studied students' LISP debugging skills and provided strong evidence that students usually debug in 4 stages: 1) understanding the system, 2) testing the system, 3) locating the error, and 4) repairing the error. They further conclude that the skills required to understand the system are not necessarily connected to the skills required to locating the error. Similarly, Fitzgerald et al. [7] explored students' perceptions of debugging, with most participants responding finding problems is more difficult than fixing them.

Beller et al. [2] studied the professionals' usage of print statements. Beller's study pointed out that hands-on debugging education in academia needs to be strengthened, as industry professionals lack knowledge of advanced debugging techniques. Based on Beller's study, our study examines the barriers to accessing debuggers in academia and how some students are successful in adapting debuggers to address the issues that were brought up in Beller's study.

Although Whalley et al. [19] did research in academia, the participants in Whalley's study were exclusively recruited from a CS2 class, which precedes the introduction to data structures. While the study offers valuable insights, its applicability to upperclassmen students dealing with intricate programming assignments involving concurrency, systems programming, and industry internship experience remains uncertain. Our study attempts to incorporate various perspectives on debugging from all levels in college.

Elliot Soloway et al. [17] studied the programming processes used and the types of and the reasons for the bugs that the introductory programming students generated. They categorized the bugs produced by novice programmers that were non-syntactic in nature. Perkins and Martin [14] studied how high school BASIC programmers think and tried to understand how they introduce errors, observing that students' difficulty in programming, including finding and removing bugs, was related to their fragile knowledge of debugging. Finally, Murphy et al. [12] explored debugging strategies for novice programmers, reporting that participants often used print statement debugging. Our study aims to advance this work to understand what prevents students from using debuggers when resolving programming bugs and what factors influence their decision-making on debugging approaches.

III. Methodology

To understand how students perceive debuggers, we sought to answer the following research questions:

RQ1: What barriers prevent programmers from using debuggers?

RQ2: How do different debugging approaches help programmers resolve bugs?

RQ3: What factors influences programmers' choice to use debuggers or print statements?

RQ4: How does using debuggers improve programming experiences?

A. Survey

We distributed a survey to Computer Science students at the authors' primary institution. The responses were all received between October 2022 and November 2022. We designed the survey to obtain examples of how respondents approach debuggers, or what prevents them from using debugging tools. We also included questions asking about debugger usage experience, behavior, limitations, familiarity, and education.

B. Survey Analysis

After collecting the survey, our next task was to extract the essence of responses. To do that, we used an **open coding** [16] approach. In open coding, a rater examines and synthesizes patterns within unstructured text. There were five open ended question in the survey and we used this approach on the responses of these questions. The first three authors of the paper individually read the responses of each open ended questions and then categorized them. Then those authors sat together to merge the categories. For the disagreements, three of them discussed together and came to a conclusion. By following these, we categorized the open-ended questions.

C. Survey Participants

We were able to recruit 73 students from diverse levels, including sophomore ($n = 2$), junior ($n = 28$), senior ($n = 15$), masters ($n = 16$), and doctoral ($n = 12$) students. Participants were familiar with the following debugger concepts: Step over (59), Step into (62), Force step into (12), step out (44), drop frame (6), run to cursor (14). Debuggers mentioned by the participants include GDB (7), Python debugger (2), Eclipse (2), Unity debugger (1), and C debugger (1). On average, students had 4.09 years of general programming experience. Due to the nature of the questions, students were not required to respond to all the questions in the survey. On the percentage calculations, we excluded participants who did not respond to questions for a more accurate result.

IV. Result

A. RQ1: Debugger Usage Barriers

The main barrier preventing debugger usage is the amount of effort needed to adopt and use debuggers ($15/39, 38\%$). Sample responses include "*Besides, debuggers seems to take much more time to find out the bugs due to its step by step debug ways*" (P15). In general, participants believed using debuggers is harder than manually inserting print statements for debugging. Many respondents also responded that they never learned how to use a debugger, especially in a formal classroom setting ($11/39, 28\%$). Even when debuggers were taught in the classroom, participants mentioned that their experience with them was limited to a tutorial during one week with only one or two exercises. Also, oftentimes students were taught how to use a command-line based debuggers when they mostly use an IDE for programming, which has a graphical user interface.

B. RQ2: Bug Resolution

Some participants ($17/48, 35\%$) weren't able to identify a circumstance where a debugger was not able to resolve a bug that they faced. Even when they have previously experienced being unable to debug their code with debuggers, they were not able to identify a case. Some programmers have even responded that spending sufficient time stepping through the code line-by-line would solve all problems. For participants who were not able to resolve a bug using a debugger, the main reason was that the tools were not able to provide enough information that the programmers needed ($6/48, 13\%$). Examples include being unable to look at some of the values that some variables have, segmentation faults, difficulty comprehending the results from the debugger, etc.

C. RQ3: Debugging Approach Decisions

To answer this research question, we wanted to gain insight into what encourages students to choose debuggers or print statements to complete debugging tasks.

1) Debuggers: The main reason participants provided for selecting debuggers is insufficient information provided by print statements. 33% $(15/45)$ of participants responded that in some cases print statements do not provide enough helpful information besides the value contained in the variable. Through this, we were able to observe that programmers are not able to use print statements to print other information that debuggers can provide, such as the address storing a variable or thread information.

We noticed that the more programming experience participants had, the more likely it was for them to use debuggers. For example, only 9% (1/11) of the students with two years of programming experience indicated that they frequently use debuggers, while more than half (52%, 14/27) of the students with five or more years of programming experience indicated that they use debuggers frequently.

Participants $(9/45, 20\%)$ were also unlikely to adopt print statement debugging for that involve system-level issues, such as I/O, segmentation faults, and memory allocation issues. Programmers responded that print statements were unhelpful in those contexts, and features that debuggers can provide, such as backtrace, help users find where a bug occurred.

2) Print Statements: Related to decisions for using debuggers, the most popular response determining print statement debugging was a lack of complexity in problems $(33/64, 52\%)$. Participants responded that they use print statements when they face errors that can easily be explained or are obvious. When it becomes more complex, such as code involving loops, hash tables, or multi-threading, then they elect to use debuggers. Another key disadvantage for debuggers is *platform dependence*. Several participants $(5/64, 8\%)$ noted they decide whether or not to use debuggers depending on their familiarity with the IDE or programming language they are using. Programmers use different IDEs depending on the programming language that they are working in, however coding environments differ in their implementation of debuggers.

Lastly, 12 participants $(12/64, 19\%)$ responded that they *use print statements first* before attempting to use the debugger. If print statement debugging goes nowhere or becomes more complex than expected, then they use the debugger.

D. RQ4: Programming Improvements

Finally, we aim to understand how debuggers improve programming experiences when adopted. The main benefits provided by participants are enhanced efficiency $(16/52, 31\%)$ compared to other debugging methods and productivity, $(10/52, 19\%)$ expanding the capability to solve bugs. Participants mentioned that they were able to resolve more bugs in a shorter frame of time after adopting debuggers. They also appreciated additional information that debuggers provide, such as memory values, helping identify bugs faster and easier. Programmers also mentioned debuggers provide a more robust experience to increase understanding of the underlying issues. Debuggers also improved experiences for other coding tasks, for instance, P51 noted a debugger *"helps a lot with testing"*.

V. Discussion

Overall, we found out the main barrier to debugger usage is the amount of time and effort needed to learn how to use them effectively. Participants also rarely encounter them in CS curricula. Further, our results suggest that the more complex the code is, the more likely the users are to use a debugger instead of print statements. However, participants tend to use print statements unless there is a substantial reason to switch to using a debugger, even if a debugger can solve the solution more effectively. Participants with debugger experience noted that it improved their efficiency, productivity, and overall programming experiences. Thus, we provide guidelines to motivate designs of future automated debugging tools and encourage the incorporation of debugger usage in CS courses.

A. Debuggers

1) Problem Difficulty: We observed that when people face more challenging programming problems, such as memory-related errors and segmentation faults, debuggers are mostly helpful. However, for simpler debugging tasks, participants opted for print statements. For example, one response was *"Yes, a lot of my problems when using c were memory allocation issues and print statements did not help with that at all."* (P31). Another response was *"I found that my program was segfaulting and not properly displaying the print statements"*. This indicates programmers seem to evaluate how complex a bug in the code is before deciding whether to use the debugger.

There are a number of opportunities to use debugging features to deal with memory problems. However, in less complex cases, participants did not seek help using debuggers. Oftentimes, students' solutions to complex problems without debuggers is copying and pasting the error message online to websites such as Stack Overflow[1] to see how other programmers that faced the same problem tried to solve the issue. This "crowd debugging" is also frequently used by novice and expert programmers [3]. Thus, integrating debuggers with Stack Overflow posts or other relevant information can save developers time from opening up a web browser and searching.

2) Advanced Coding Concepts: More advanced coding concepts, such as recursion and loops, also led participants to use debuggers instead of print statements. For instance, one respondent mentioned that *"I always used print statement or search online. I would say I'll use debugger when there is some error caused by looping issue"* (P15). Because it would be very difficult to debug via print statements when there are hundreds of lines printed in the console, developers use debuggers to avoid this issue. Additionally, complex data structures also led developers to use debuggers. For example, one participant responded that *"If i have a hash table or something Id definitely use a debugger but if i have only a few variables being updated i can print them out"* (P30). This motivates the need for debugging tools to help developers understand the complex programming concepts while debugging, through techniques such as visualizations to improve

[1]https://stackoverflow.com

user understanding [8]. We could also use this information to nudge students to use debuggers more often by automating customized recommendations for specific debugging features based on the data structures and programming paradigms included in their code.

3) Platform Independence: One challenge reported in our survey is that participants are hesitant to learn debuggers for different languages and IDEs. For example, one response was *"I have used the python debugger, which I have found extremely useful because of the ability to easily interact w/ the current program state via the interpreter. For JS, I feel like it could maybe be useful, but printing objects to the console is quick, easy and effective."* (P5). Another noted *"printing statement works better for me since I don't really like to get used to debugger for every new IDE environment."* (P15).

Varying interfaces and functionality in different debugging tools makes potential users disinterested in using debuggers, hence the prevalence of print statements. To resolve this issue, we posit debuggers with similar user interfaces and interactions across multiple IDEs are ideal. Although, we do recognize that consistent interfaces would be hard to be applied to all IDE plugins available for developers, we believe that this can be implemented in most common programming languages, such as Java, Python, and C.

B. Education

1) Debugging education can play an important role: Most students that do not use debuggers responded that they never had the opportunity to learn how to use debuggers or learned too briefly. A plausible factor contributing to this phenomenon could be attributed to a potential lack of experience using complex debuggers found in modern IDEs among CS instructors. Prior work also suggests debugging tool usage is fragile and often neglected in CS education [15]. For instance, many new programmers struggle in the first few months of new jobs due to a lack of experience in various software engineering tasks, including debugging. [1]. Additionally, a glance at popular programming-related sites also seem to confirm this problem [6], [18], and a lot of the questions asked and problems faced by students can be solved with the proper use of debuggers [11]. We received responses such as *"We weren't really taught how to use it and it was either the IDE will do it automatically or good luck"* (P12).

Moreover, participants who did receive instruction on debuggers noted the irrelevance of lessons, such as learning to use command-line debuggers like gdb[2] while most development is done in IDEs. Due to limited exposure to debuggers, students may be unaware of the benefits of automated debugging tools. Further research is needed to explore how to teach debuggers effectively in a classroom setting, which can be as simple as increasing the hours spent on teaching debuggers and having more exercises where students use debuggers. Further, educators can create coding assignments complex enough to make print statement debugging extremely difficult, encouraging students to use debuggers. Finally, introducing a course dedicated to debugging techniques would also be useful for teaching up-to-date debugging practices in industry.

2) External Resources: Almost all students that use debuggers responded positively about their debugger usage. Some students responded with passion, such as *"10000% debugger is so important, If you dont use it, you tend to just pull out your hairs for hours and coding becomes a very frustrating experience"* (P19). Although the main barrier to debugger usage is that they can be hard to learn, several participants noted it is worth learning. Many survey respondents noted using external resources to learn how to use debuggers, such as online blogs and YouTube videos. This motivates new research directions to explore how to make debugging learning experiences easier for students and instructors, to prevent debugger avoidance. Outside of the classroom, content from tutorials and other online resources can be evaluated and improved to help programmers learn and practice using debuggers.

VI. Limitations and Future Work

We have only explored perceptions of debugger usage with a limited set of participants from one university. While we attempted to recruit a diverse sample of participants in terms of professional and academic experiences, they may not represent the debugging practices of all programmers at varying levels or CS students at other institutions. Future work should conduct more extensive recruiting from a broader range of participants to increase the validity of our results.

In the future, we hope this work motivates new techniques to improve the adoption of debuggers. First, we aim to explore investigating more usable debugging tools to overcome challenges expressed by participants, such as platform dependence and problem difficulty. Moreover, we plan to explore methods to help potential users learn about debugging tools more effectively. This includes investigating updates to CS curricula, coursework activities focused on debugging, enhancements to tutorials and resources to train users, and tools to automate recommendations for debugging tools and techniques.

VII. Conclusion

In this paper, we surveyed participants to understand the barriers that keep students from learning skillful usage of debuggers. Responses suggest that CS courses do not usually focus on effective and relevant debugger usage, leaving students with a lack of knowledge in this area. We also see that students struggle with complex debugger UIs, especially when they vary between IDEs, which intimidates them to an extent when it comes to the initial learning curve. However, students are more open to using debuggers when solving more complex data structure problems. Our results provide implications for improving debugging tools and CS education to reduce the effort barrier preventing programmers from using debuggers to find and fix errors in their code.

[2]https://www.sourceware.org/gdb/

References

[1] Andrew Begel and Beth Simon. Novice software developers, all over again. In *Proceedings of the fourth international workshop on computing education research*, pages 3–14, 2008.

[2] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*, pages 572–583, 2018.

[3] Fuxiang Chen and Sunghun Kim. Crowd debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 320–332, New York, NY, USA, 2015. Association for Computing Machinery.

[4] Ophelia C Chesley, Xiaoxia Ren, and Barbara G Ryder. Crisp: A debugging tool for java programs. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 401–410. IEEE, 2005.

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

[6] dizzyflames. Is debugging with print statements bad?, Jan 2019.

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

[8] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. Vida: Visual interactive debugging. In *2009 IEEE 31st International Conference on Software Engineering*, pages 583–586, 2009.

[9] Bryce Ikeda and Daniel Szafir. Advancing the design of visual debugging tools for roboticists. In *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 195–204, 2022.

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

[11] Kelly Loougheed. 10 best practices for helping students debug their code, July 2019.

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

[13] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[14] David N. Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. 1985.

[15] David N Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 213–229, 1986.

[16] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. Sage, 2015.

[17] Cutler B Spohrer, Draper S and Elliot Soloway. Bug catalogue: I (technical report no. 286). 1983.

[18] user113476. Why aren't students taught to use a debugger?, Jan 2010.

[19] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. Novice reflections on debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 73–79, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Shaochun Xu and Václav Rajlich. Cognitive process during program debugging. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, pages 176–182. IEEE, 2004.

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