2023 Annual Conference & Exposition
Baltimore Convention Center, MD | June 25 - 28, 2023
The Harbor of Engineering
Education for 130 Years
ASEE
Paper ID #39230

Preference for debugging strategies and debugging tools and their relationship with course achievement: Preliminary results of a study involving novice programmers.

Dr. Laura Melissa Cruz Castro, University of Florida

Laura Melissa Cruz Castro is an instructional assistant professor in the engineering education department at the University of Florida.

Jenny Patricia Quintana-Cifuentes, University of Louisiana at Monroe

Jenny Quintana is an assistant professor at the University of Louisiana Monroe. Dr, Quintana has a Ph.D. in Engineering Education from Purdue University and two master's degrees from Purdue University. One is in Technology leadership and innovation, and the other is environmental and ecology engineering. Dr. Quintana completed her undergraduate studies in Technological Design in Bogota, Colombia.

Akash Kumar

Preference for debugging strategies and debugging tools and their relationship with course achievement - preliminary results of a study involving novice programmers.

Abstract

It is estimated that 35-50% of the time spent building solutions is spent debugging software. Despite the importance of debugging in software development, how students debug, and the strategies and reasoning students use for debugging software are still unclear. This work in progress will present preliminary results on students' preferred debugging strategies and compare them with their learning gains during a programming course. We focus on answering the following questions: a) "*Is there a difference between students' preference for debugging strategies and their course achievements?*"; b) *"Is there a relationship between software debugging tools and the conceptual understanding of debugging strategies?"*

This study was conducted during Fall of 2022 in a 16-week programming fundamentals II course at a large public southwestern university. This semester, 328 students enrolled from various engineering and computer science majors. The data was gathered from a debugging assignment, which is an open-ended questionnaire. The open-ended questionnaire aims to uncover students' thought processes when helping others to debug their code and students' debugging strategies. In addition, a coding book was developed to capture students' utterances and classify them into Metzger's debugging strategies: a) incremental development, b) program slicing, c) sanity checks, d) error variables for controlling behavior, e) cause elimination methods, f) turning debugging code on and off and g) traceback. Preliminary results include the finalized codebook established by four coding sessions and two researchers who are also experienced programmers and the descriptive statistics from ten students coded with the final version of the codebook.

Through the analysis of the open-ended responses of ten students, our study sought to identify the debugging strategies that students use concerning their learning gains. From the information gathered, most students were able to verbalize two strategies. Sanity Checks Low Level (SCL), and Sanity Checks High Level (SCH). In addition, it was possible to observe that students whose approach included SCH had higher academic achievement in the course. In contrast, students reflecting a preference for SCL had lower performance in the course. In addition, it was found that students using debugging tools expressed a wider range of debugging strategies, and all of them were able to express their use of debugging strategies that are further facilitated by debugging tools.

Introduction

In recent years, focusing on preparing students for big data and artificial intelligence has led to a progressive interest in developing students' programming skills. A big part of artificial intelligence and big data is concerned with software development, which often relies on effective debugging strategies. Debugging is a process in which a failure is observed, identified, and removed [1], and it is estimated that 35-50% of the time spent building solutions is debugging [2].

Different tools and strategies are believed to help programmers debug programs. Nowadays, debugging tools exist within Integrated Developing Environments (IDEs) in addition to other specific scenarios (e.g., [3]–[6]). However, identifying and resolving failures in software is still challenging [7]. One reason for being a challenging process is that debugging is a form of troubleshooting that is inherently difficult for human beings [8]. This challenge adds to the debugging process complexity when accounting for the nature of bugs. Furthermore, bugs are created by programmers and implicitly in-built into the code itself [7]; in other words, bugs are technically created by programmers. Therefore, the debugging process requires identifying when bugs occur, subsequent dependencies that can propagate the error, and finding the best path to solve it [9], [10].

Debugging skills have been explored in higher education. For example, some researchers have created forms of instructional scaffolding that help novice students to develop debugging skills [11], [12]. In addition, several studies have focused on comparing expert and novice programmers debugging skills to understand "*What do experts know that we can transfer to novice programmers?"* focusing on developing tools and strategies to help students. However, little attention has been paid to what novice students know. On that account, arguably while developing strategies to help students is relevant and comparing novices with experts helps us understand differences between these two groups, novices' debugging skills and strategies used are often not fully explored.

The lack of exploration of novice programmers' debugging skills has led to learning and teaching strategies that do not consider or understand how students deal with debugging processes in software development. For instance, it is assumed that students learn debugging by having experience with debugging [13]. However, a study by Whalley and colleagues revealed that students' reflections on their experiences with debugging tend to be negative [14]. In this study, students expressed that exploring strategies such as print statements frequently will make them miss the program's general idea, forcing them not to follow a methodological approach [14].

Although debugging is a challenging task, it is also an essential skill that students must master to acquire other computational thinking skills [15]. Consequently, exploration of students' debugging skills is essential to develop teaching and learning strategies that fully explode their already-in-place preferences and knowledge. Understanding students' preferences and their natural approaches to the debugging process is critical to advancing computing education. Furthermore, achieving this goal will help the computing community understand intuitive approaches students use while dealing with the complex debugging process to adapt tools and learning experiences further. To this end, two research questions are explored in this work-in-progress paper: RQ1: "*Is there a difference between students' preference for debugging strategies and their course achievements?*" and RQ2: *Is there a relationship between the use of software debugging tools and the preferences for debugging strategies?*

Conceptual framework:

This study used the Debugging by thinking conceptual framework proposed by Metzger in 2003[1]. Metzger uses a multidisciplinary approach to tackle the understanding of debugging. Metzger uses the term "Debugging by thinking " to refer to the expert programmers' approach to debugging. Metzger's proposal of debugging by thinking consists in approaching the debugging task with an explicit methodology, seeking problem-solving methods from multiple disciplines, and with awareness of the assumptions of the solution and probable mistakes involved in the development [1]. Metzger argues that although it is believed that debugging is learned by trial and error, oftentimes, this results in programmers learning "tricks" to resolve problems with their program without a clear understanding of the cause or a well-developed methodology to solve other debugging problems. Therefore, a constructive, well-rounded, and multidisciplinary approach is required to become an expert debugger. In the attempt to define debugging by thinking, Metzger proposes six debugging strategies or ways of thinking about a debugging problem: Incremental Program Development, Program Slicing, Sanity Checks, Catch and Save Error, Cause Elimination Method, and Test Cases. These strategies inform our study to capture a broad range of debugging preferences and expertise levels that we expect from our students. In Table 1, the definition of each strategy from Metzger's framework is provided.

Table 1
Definition of the six debugging strategies proposed by Metzger[1].

Strategy	Definition
Incremental program development	This strategy consists of the smallest portions of the code that are tested and debugged before incremental changes are made and added to the overall program. This approach helps reduce the debugging process's complexity and ensures that each portion of the code works as intended before moving on to the next.
Program slicing	This strategy consists of the programmer focusing on a specific portion of their code (by commenting out or isolating a section) to identify any errors or problematic behavior in that section.
Sanity checks	A type of error checking where the programmer inserts check throughout the code to validate low-level assumptions such as the type of variables, syntax, values of parameters, and object definitions. The goal of sanity checks is to identify problems at an early stage of development quickly.
Catch and Save Error	This technique refers to the practice of catching errors or exceptions that occur during the execution of a program and storing or printing the error message. This information can then be used to identify the source of the error and diagnose the issue. In addition, by capturing and preserving the error, developers can better understand the cause of the problem and take steps to correct it.
Cause elimination method:	A debugging technique where the programmer first identifies the source of an error then generates a list of

	possible causes of the error and eliminates each cause one by one until the root cause of the error is found.
Test case	A test case is a set of steps, inputs, and expected outcomes used to test a specific aspect of a program. Test cases are designed to verify that the program behaves correctly and performs its intended functions.

Literature Review

What has been done around debugging in higher education?

In higher education, debugging literature has identified several key components necessary for students to learn how to debug effectively. Ko and Myers in 2005 state that these components include the ability to understand code, recognize common error messages and their causes, and plan a debugging strategy[16]. In addition, empirical studies in higher education have also compared the debugging strategies of diverse groups of programmers, such as novice and expert programmers. For instance, novice programmers tend to rely on a "trial and error" approach. In contrast, expert programmers use a more systematic approach that involves analyzing the problem, generating hypotheses, and testing the hypotheses [17]. Another study found that expert programmers were likelier to use debugging tools, such as debuggers and error tracebacks, than novice programmers [18]. While another study found that students taught debugging strategies through problem-based learning had significantly better debugging skills than those taught through traditional lectures [19].

In addition to the comparisons between expert and novice programmers, multiple teaching interventions have been aimed at improving students' debugging skills in higher education. One such intervention is debugging exercises, effectively improving students' debugging skills [[19]. Another teaching tool that has been used previously corresponds to debugging games. Debugging games have taught debugging skills effectively [20]. Another innovative approach is using virtual debugging tutors, which are computer-based systems that provide personalized feedback and guidance to students as they work through debugging problems [[18]. These virtual tutors improve students' debugging skills and performance in computer science courses [21]. Finally, exposure to systematic debugging has also been tested, finding relationships between systematic debugging exposure and students' self-efficacy and effective debugging ([22], [23]).

Debugging and students' performance

Previous research has established the complexity and multiple factors that influence students debugging performance. To date, several studies have focused on how the program error message influences students' skills and strategies to debug [24], the time novice students take to debug a problem by using counting error compilers [13], identifying how visual attention could also impact students debugging performances [25] and the type of high or lower achievers influence students' strategies and performance on debugging [13], [26].

Studies have shown that students spend less time and effort while debugging tools that provide more detailed feedback on errors [24] or mirror tools that help students to become aware of the errors [25]. Research literature has also revealed that while high-performance students focus more on the program's logic flow while debugging, low-performance students focus more on accessing previous information entered into the program to compute a value [27]. These findings complement prior research that indicates how novice students rely on trial and error rather than understanding a program's logic while debugging a program [28], [29].

Methods:

Setting and participants:

This study was conducted in a large public southeastern institution programming fundamentals course. During Fall 2022, there were 328 students enrolled in the course. The course is a second part of programming fundamentals courses and focuses on using C++. This 16-week course is divided into roughly ten weekly assignments, two major exams, two intermediate projects, and one final project. Both assignments and lectures revolve around the following goals (1) Build and execute C++ programs from the command-line and an IDE (2) Demonstrate an ability to solve large programming problems (3) Examine the uses of dynamic memory allocation, pointer, and template to create the creation of memory-efficient data structures such as linked lists, stacks, and queues. In order to answer our research questions, the proxy of students' performance was the midterm examination. The midterm exam contained a total of 17 conceptual questions and 18 programming questions. Conceptual questions were multiple choice, and programming questions offered students code snippets that could contain errors or code snippets in which they would need to provide the program's output.

Data collection

The instrument for gathering our data was a survey with a total of 15 questions. Five of the questions focused on eliciting students' programming experiences, three questions focused on students' demographics, and seven questions targeted to exploring students' debugging strategies. This survey was distributed during week 12 as an assignment. Three hundred twenty-three students submitted the assignment; however, after cleaning duplicate students and empty responses, there was a total of 302 completed assignments.

This working-in-progress study uses the student's responses to question 13. Question 13 is as follows:

> *"To complete this part of the assignment, you must write 300 words or more. Suppose you would have to help a person to debug their code. What would be the process that you would use to help this person? Describe the process with specific steps and the questions that you would ask." (Q13)*

The data from each student's response was organized into discourse segments. Each discourse segment was divided based on students completing an idea. Researchers agreed on each discourse segment to be further coded. An example of our data before and after the segments is presented in Table 2.

Table 2
Example of the discourse segments for one student

Student’s answer excerpt without discourse segment	Discourse segments
“The first thing I would check would be [platform name]. As most of our submissions are through this website it can be useful to weed out obvious mistakes. Secondly, I would ask them "What step do you believe there could be a potential problem in?". If their answer is "IDK" then I would explain to them an overhaul of the project. If they understood all aspects and didn't have a conflict with understanding my next step would be to look over their code.” (Student 1)	Discourse segment 1: “*The first thing I would check would be [platform name]. As most of our submissions are through this website it can be useful to weed out obvious mistakes.”* Discourse segment 2: “*Secondly, I would ask them "What step do you believe there could be a potential problem in?". If their answer is "IDK" then I would explain to them an overhaul of the project. If they understood all aspects and didn't have a conflict with understanding my next step would be to look over their code.”*

Coding book:

A content analysis was used to classify each student discourse into a category [30]. Next, we developed a coding book protocol using the conceptual framework. The initial coding book contained seven categories corresponding to debugging by thinking strategies [31]. These categories included: linear incremental debugging, program slicing, sanity checks, catch and save errors, cause elimination method, test cases, and no code.

Two researchers performed a content analysis of the students' discourses. A total of four rounds of coding were performed using the most updated coding book each time. The first coding round corresponded to coding one student with the codebook’s initial version. Each researcher verified their code for this student, and we finished this round after achieving consensus on the strategy for all the discourses of this student. In order to achieve consensus, some of the codebook definitions for each category were changed to align with data and researchers’ observations. For example, two changes made to the coding book during our coding rounds were eliminating traceback techniques and the change of the linear incremental development strategy. First, the traceback technique was eliminated because this strategy involved the interaction between multiple strategies, such as the cause elimination method and program slicing, causing disagreement among the researchers. Second, incremental program development changed to become linear incremental debugging. In this case, students were asked to help others debug; therefore, incremental development would not be present in the context of the question. Linear Incremental Debugging (LID) encompassed the incremental debugging portion of students' strategies.

For the second coding round, the two researchers were tasked with coding three new students and with a second version of the codebook. After coding individually, we met to discuss the codes for each student. We redefined our coding book until an agreement was reached for each student. From this round, some other changes were produced. For example, sanity checks were divided into two categories to reflect the higher-level approach that some students reflected in their answers and still needed to be captured by the codebook. Because of a high percentage of disagreements with the first students, it was decided to return to the two coded students and revise the code with the new codebook. The third round then consisted of a meeting to check the two remaining discourses; after agreeing with the code for these students and having a smaller percentage of disagreement, each researcher coded ten students individually.

The fourth and final coding round consisted of checking our codes' reliability on the ten students. The intraclass correlation coefficients (ICCs) proposed by Cohen and Doveh were calculated using the coded discourses corresponding to the new ten students[32]. For this final coding round, the ICC index among the two coders reached 0.81. All codes were verified, and all students coding reached 100% agreement among researchers. No further changes to the codebook were made, which was considered the final version (see Table 3).

Table 3.
Final version of the codebook

Code name	Definition	Student quotes examples
Linear Incremental Debugging (LID)	The student indicates that the smallest portions of code will be tested linearly until something is erroneous. For example, it could move sequentially forward or backward.	“Then, I would ask them to run these chunks, one at a time, from top to bottom, and see if any of the chunks produce an error.”
Program Slicing (PS)	The student indicates that they will focus on a portion of their code (e.g., comment out certain functions) to identify erroneous behavior.	“The next thing that I would do to help is asked, "can you separate the blocks of code based upon functionality?" This would allow them to chunk their code and allow them to much more easily test it.”
Sanity checks Low-level. (SCL)	The student indicates that they use checks throughout their code (in the form of conditionals and print statements) on low-level assumptions, such as type of variables, syntaxis, values of parameters, and definitions for the objects that exist. In addition, they may also indicate the use of the IDE console errors/warning to find a starting point in debugging without catching the errors.	“The error should give a line/function it happens in, which can narrow down the search.”
Sanity checks high-level. (SCH)	The student indicates that they check the design, logic, objective, purpose, or other high-level assumptions of the program.	“The very first thing that I would ask this person to do is to create a diagram of the functionality of their program, so that they can see for themselves how it is supposed to run.”

Catch and Save Error (CSE)	The student indicates that they store/print (throughout the program) error generated by the program.	"… you could use error handling tools like try and catch statements. You could use these by surrounding the error with these statements and then seeing if it's printing out the error."
Cause Elimination Method (CEM)	The student indicates that they know where the error is coming from. Then, they analyse the possible contributors to such errors. After analysing, they develop a list (or they mention a sequential or rationale-based way) of possible causes of the error [at the symptom level (part of the software that is not working)] and then conduct tests to eliminate each of the possible causes.	"The third method would be commenting things out because if there's an error in a function you can comment out the whole function and then comment line by line back in the function and seeing where the error comes into play."
Test cases (TC)	The student indicates that they will build test cases to test specific functionality of the program.	"If the answer is yes, then I would ask them to create test cases for those methods or classes, separate from any other part of their code. If these test cases result in incorrect functionality, I would ask them to focus on the code inside of those methods or classes and use the newly created test cases to debug them."
No coded (NC)	The researcher indicates that the discourse did not contain any debugging strategy	"To help a person out with debugging their code, first i would look at the errors given on their compile line and ask them if they know why such errors are showing."

Results:

The results presented in this work-in-progress paper correspond to ten students randomly selected from the pool of 302 students whose answers to Q13 were coded with the final version of the codebook presented in Table 3. The students indicated debugging strategy per coded discourses, their indication of use of a debugging tool, and their midterm grades were analyzed to answer the two research questions.

RQ1: Is there a difference between students' preference for debugging strategies and their course achievements?

RQ2: Is there a relationship between the use of software debugging tools and the preferences for debugging strategies?

The descriptive statistics are presented for the analyzed data, in addition to preliminary results for RQ1 and RQ2.

Descriptive statistics

The average number of discourse segments found in each student's answers to Q13 was seven, with a total of 70 discourse segments coded across all ten students. In addition, 19 discourses were coded as Sanity Checks High Level (SCH). SCH was the most frequent strategy identified in students' answers, with six students using this strategy, followed closely by Sanity Checks Low Level (SCL) with eight students. Table 4 presents the number of coded debugging strategies in each student's answers. Interestingly, Cause Elimination Method (CEM) and Test Cases (TC) were rarely found in the coded questions, with only one and two students using them, respectively. Catch and Save Error (CSE) was not found in the sample, while No coded (NC) discourse segments were frequent. NC data indicated no evidence of any use of any debugging approach.

Table 4

Number of students and discourses segments coded in each strategy.

Strategy	# Students	# Discourses
LID	6	9
PS	6	8
SCL	8	15
SCH	9	19
CSE	0	0
CEM	1	1
TC	2	2
NC	8	16

Note: CEM = Cause Elimination Method; CSE = Catch and Save Error; LID = Linear Incremental Debugging; NC = No Code; PS = Program Slicing; SCL = Sanity Check Low Level; SCH = Sanity Check High Level; TC = Test Cases

In addition, in Table 5 presents the exact sequence observed for each student including repeated strategies and discourses classified as not including any debugging strategy (NC).

Table 5

Sequences of students' coded discourses

Student ID	Coded strategy from the discourse segment sequence							
1	NC	SCH	LID	NC	PS	NC	NC		
2	SCL	SCH	PS	LID	PS	SCL	NC	NC	
3	SCH	SCL	SCL	PS	PS	SCH	TC		
4	NC	LID	SCH	NC	LID	SCH			
5	SCH	SCL	LID	SCH	LID	LID	NC		
6	SCH	SCL	SCL	LID	NC	PS	SCL	NC	TC
7	SCH	SCH	CEM	SCH	SCH	NC	NC	SCL	
8	NC	LID	SCL	SCH	NC	TC	PS		
9	SCL	PS	SCL	SCL	SCL				
10	SCH	SCH	PS	SCL	SCH				

Note: CEM = Cause Elimination Method; CSE = Catch and Save Error; LID = Linear Incremental Debugging; NC = No Code; PS = Program Slicing; SCL = Sanity Check Low Level; SCH = Sanity Check High Level; TC = Test Cases

RQ1: "*Is there a difference between students' preference for debugging strategies and their course achievements?*"

Table 6 results from the sequence of coded discourse (Table 5) when discarding NC and repeated strategies instances. From Table 6 it is possible to observe that most students were identified to prefer SCH as a first step for debugging a code. Students indicated using the SCH strategy when given their debugging task to understand the program's problem, logic, and objective. The second most common first-step strategy was SCL, mainly used by students who were trying to first clear compiler errors or syntax errors at the beginning of the process. Finally, two students indicated using LID to start the debugging process, mainly indicating their approach as starting in their first line of code and moving sequentially. In addition, at most, a student could verbally express five different strategies and a minimum of two strategies. In addition, there is a maximum grade in the exam of 92.25 points and a minimum of 56.17 points over 100 points.

Table 6

Sequences of students' coded discourses without NC or repeated strategies

Student	Exam grade	Use of a debugger	Coded strategy from the discourse sequence				
1	84.33	no	SCH	LID	PS		
2	66.67	no	SCL	SCH	PS	LID	
3	66.75	no	SCH	SCL	PS	TC	
4	75.92	no	LID	SCH			
5	56.17	yes	SCH	SCL	LID		
6	75.42	yes	SCH	SCL	LID	PS	TC
7	87.83	no	SCH	CEM	SCL		
8	92.25	yes	LID	SCL	SCH	TC	PS
9	57.67	no	SCL	PS			
10	68.83	yes	SCH	PS	SCL		

Note: CEM = Cause Elimination Method; CSE = Catch and Save Error; LID = Linear Incremental Debugging; NC = No Code; PS = Program Slicing; SCL = Sanity Check Low Level; SCH = Sanity Check High Level; TC = Test Cases

In order to answer the research first research question, a bar plot with standard error bars was created with the most frequently coded strategies, as shown in Figure 1. As it was identified in Table 4, SCH, SCL, LID, and PS were strategies that were coded in multiple students (More than two students). Therefore, these strategies were used to identify differences between the most preferred strategies by the students and their course achievements.

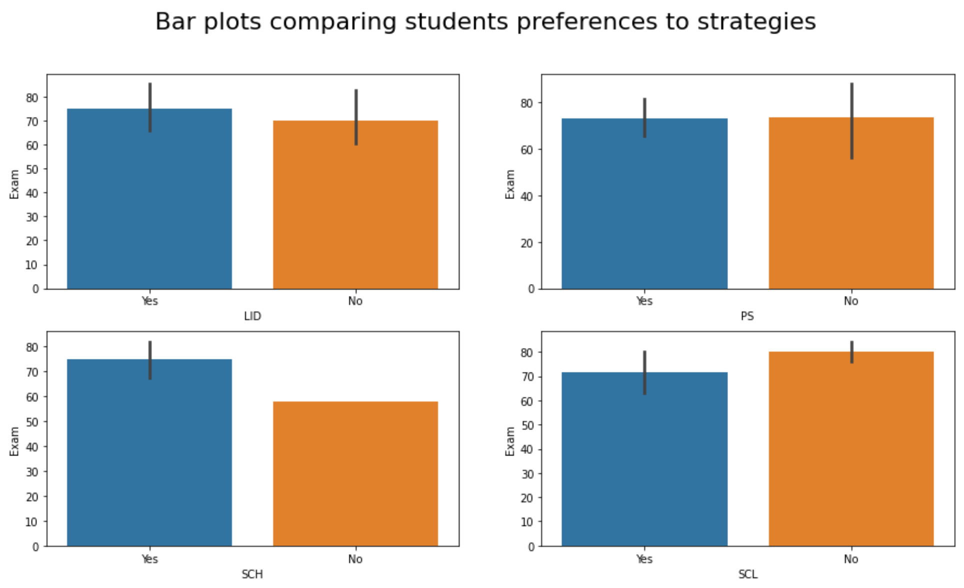

Figure 1: Midterm exam scores by at least one appearance of each debugging strategy in the coded answer to Q13 for Linear Incremental Debugging (LID), Program Slicing (PS), Sanity Check High Level (SCH), and Sanity Check Low Level (SCL).

From Figure 1, it is possible to observe that both SCH and SCL influence students' performance in the exam. However, students whose assignments reflected a preference for SCL have slightly lower grades than those whose answers did not reflect a preference for SCL. On the contrary, students that preferred SCH had higher grades on the exam. Nonetheless, it is important to note that for SCH, there was only one student whose answer needed to contain this strategy. Although error bars are not available in this case, this student has one of the lowest grades in the exam. It is also important to note that the preference for LID and PS does not influence students' performance in the exam. Nonetheless, while for PS, there is no visual change between students that reflected a preference for this strategy, for LID, there seems to be a slightly lower exam grade from students who did not.

RQ2: "*Is there a relationship between the use of software debugging tools and the preferences for debugging strategies?*"

After removing discourse segments coded as NC and repetitive strategies per student, it is possible to identify that two students' answers use five debugging strategies from the eight strategies. This pattern is seen in Student 6 and Student 8, presented in Table 6. Although these students were coded with the same five strategies, the sequence in which they appear differs. This difference could indicate that even though they prefer the same strategies, it is given different priorities. On the opposite side of the spectrum, there are two students whose answers were only coded with two different strategies, Student 4 and Student 9. In this case, both strategies differ between these two students, and both students indicated that they do not use software debugging tools.

Four of the ten students in the sample indicated a debugging tool (Table 6). To explore research RQ2, the two groups were explored separately. When evaluating the number of strategies reflected in the student's answers, it was found that the average number of strategies coded per student was different among students who preferred a debugging tool versus those who did not indicate this preference. Students who indicated the use of a debugging tool reflected, on average, the preference for four debugging strategies, while students who did not indicate the use of a debugging tool reflected an average of three debugging strategies. This result could potentially indicate that students who use a debugging tool might be able to navigate more strategies than their counterparts.

Table 7:
Students' discourse segments percentage in each strategy based on the use of debugging tool coded in each strategy.

	YES (n = 4)	NO (n = 6)
LID	75%	50%
PS	75%	67%
SCL	100%	67%
SCH	100%	83%
CSE	0%	0%
CEM	0%	17%
TC	50%	17%

CEM = Cause Elimination Method; CSE = Catch and Save Error;
LID = Linear Incremental Debugging; NC = No Code;
PS = Program Slicing; SCL = Sanity Check Low Level;
SCH = Sanity Check High Level; TC = Test Cases

The proportion of students' answers to Q13 that contained at least one discourse coded with the respective strategy is presented in Table 7. From Table 7, it is possible to observe that students using a debugging tool seem to consistently express the use of both SCL and SCH, with 100% of them reflecting the use of both strategies. In addition, there is a higher rate of students' preference for LID and PS than their counterparts. Half of the students using a debugging tool indicated they used TC, and both CSE and CEM which were strategies scarcely found in both groups.

Discussion:

This study focused on understanding the relationship between students' debugging strategy preferences and their course achievements, particularly their midterm exam performance. Preliminary findings of this working-in-progress study indicate that students that indicated both Sanity Check High Level (SCH) and Sanity Check Low Level (SCL) as their debugging strategies showed a correlation with the midterm exam grade. In contrast, SCH seems to positively affect performance, while SCL may negatively affect students' performance. The student preference for these strategies and their effects on performance supports Whalley and colleagues' findings [14]. According to Whalley et al., students struggle with debugging tasks when they try to use low-level strategies (such as using print statements) and miss the program's objective. This finding also aligns with the understanding that novice programmers might rely on trial and error rather than the program's logic [28], [29]. Therefore, amidst further exploration of this correlation, it would be worthwhile to create course strategies that help students focus on debugging from a higher-level perspective.

Other findings from this study suggest that students who indicated using a software debugging tool reflected in their answers a more diverse set of strategies than their counterparts. It was also noted that all students indicating the use of a debugging tool were able to express the use of Linear Incremental Debugging (LID), Sanity Checks Low Level (SCL), and Program Slicing

(PS), which are strategies that debugger tools make easier to operationalize. This result was interesting, as one of our initial hypotheses was that students who use debugging tools might need help verbalizing the strategies as they might not be conscious about their use due to their reliance on the tool. Although one explanation for these results could be that debugging tools help students conceptualize and understand debugging strategies, another possibility exists. It has been found before that more experienced programmers are more prone to use debugging tools [18]. Therefore, these students have used these strategies before and, because of this manual use, can verbalize the use of the debugging strategies.

It is worth noting that while most students in our sample verbalized SCH and SCL strategies, they were unable to go a step forward, such as generate a hypothesis or test a hypothesis, which characterizes debugging for expert programmers [17]. In addition, all students in this study were able to develop a plan involving more than one debugging strategy, which characterizes expert programmers [16].

Finally, the importance of debugging for software development and the preparedness of our students for the workplace is of foremost importance [4]–[6], [33]. While we explore ways to help students acquire expert programming levels as early as possible, acknowledging the importance of debugging in teaching others, more complex programming skills are critical [15]. Individual practice may not be enough to acquire debugging skills and could potentially frustrate students [14]. Therefore, it is necessary to understand the strategies that students are using and leverage what they bring to create a scaffolding process in which we introduce strategies closely related to those they currently use [11], [12]. In addition, amidst further results, it would be relevant to find specific moments and ways in which tools (such as debugging tools) are introduced, keeping in mind that students understand various debugging strategies appropriate for their level of expertise.

Strengths and limitations of this study

Some of the limitations of this study are concerned with the sample size and the assignment time. For this study, only ten students were coded; therefore, although they shed light and prepared us to investigate the research questions further, the generalizability of the results is jeopardized. In addition, while the assignment is released in a week in which students are working in parallel on a programming assignment that requires them to debug, it is possible that during their debugging assignment, they have to recall information from previous experiences.

Some of the strengths of this study relate to the question asked and the coding process. For instance, novice programmers might find it difficult to express the use of certain strategies when asked directly about their use; nonetheless, with the open question, students could express their use and understanding of the debugging process without using specific words to address the strategies. In addition, the coding process involved two researchers, who are experienced programmers and have worked closely with students in debugging assignments, and a total of the fourth rounds of coding, which makes the codebook one of the most important outcomes in this preliminary work.

References

[1] R. C. Metzger, “Debugging heuristics,” in *Debugging by thinking: A multidisciplinary approach*, 2003, pp. 201–219.

[2] D. H. O’Dell, “The Debugging Mindset,” *Queue*, vol. 15, no. 1, pp. 71–90, Feb. 2017, doi: 10.1145/3055301.3068754.

[3] J. Bennedsen and C. Schulte, “BlueJ Visual Debugger for Learning the Execution of Object-Oriented Programs?,” *ACM Transactions on Computing Education*, vol. 10, no. 2, pp. 1–22, Jun. 2010, doi: 10.1145/1789934.1789938.

[4] P. Ardimento, M. L. Bernardi, M. Cimitile, and G. de Ruvo, “Reusing bugged source code to support novice programmers in debugging tasks,” *ACM Transactions on Computing Education*, vol. 20, no. 1, pp. 2–24, Nov. 2019, doi: 10.1145/3355616.

[5] H. L. Nguyen, N. Nassar, T. Kehrer, and L. Grunske, “MoFuzz,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, Dec. 2020, pp. 1103–1115. doi: 10.1145/3324884.3416668.

[6] J.-H. Ji, G. Woo, H.-B. Park, and J.-S. Park, “Design and Implementation of Retargetable Software Debugger Based on GDB,” in *2008 Third International Conference on Convergence and Hybrid Information Technology*, Nov. 2008, pp. 737–740. doi: 10.1109/ICCIT.2008.268.

[7] A. Zeller, “Where do bugs come from?,” *Electron Notes Theor Comput Sci*, vol. 174, pp. 55–59, May 2007, doi: 10.1016/j.entcs.2006.12.029.

[8] N. M. Morris and W. B. Rouse, “Review and evaluation of empirical research in troubleshooting,” Oct. 1985.

[9] J. Moondanos, “From error to error: Logic debugging in the many-core era,” *Electron Notes Theor Comput Sci*, vol. 174, pp. 3–7, May 2007, doi: 10.1016/j.entcs.2006.12.035.

[10] C. Li, E. Chan, P. Denny, A. Luxton-Reilly, and E. Tempero, “Towards a framework for teaching debugging,” in *ACM International Conference Proceeding Series*, Jan. 2019, pp. 79–86. doi: 10.1145/3286960.3286970.

[11] B. Zhong and Q. Si, “Troubleshooting to learn via scaffolds: Effect on students’ ability and cognitive load in a robotics course,” *Journal of Educational Computing Research*, vol. 59, no. 1, pp. 95–118, 2021, doi: 10.1177/0735633120951871.

[12] L. Zheng, Y. Zhen, J. Niu, and L. Zhong, “An exploratory study on fade-in versus fade-out scaffolding for novice programmers in online collaborative programming settings,” *J Comput High Educ*, vol. 34, no. 2, pp. 489–516, Aug. 2022, doi: 10.1007/s12528-021-09307-w.

[13] M. Ahmadzadeh, D. Elliman, and C. Higgins, “The impact of improving debugging skill on programming ability,” *Innovation in Teaching and Learning in Information and Computer Sciences*, vol. 6, no. 4, pp. 72–87, Oct. 2007, doi: 10.11120/ital.2007.06040072.

[14] J. Whalley, A. Settle, and A. Luxton-Reilly, “Novice reflections on debugging,” in *SIGCSE 2021 - Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, Mar. 2021, pp. 73–79. doi: 10.1145/3408877.3432374.

[15] L. M. Cruz Castro, A. J. Magana, K. A. Douglas, and M. Boutin, “Analyzing Students’ Computational Thinking Practices in a First-Year Engineering Course,” *IEEE Access*, vol. 9, pp. 33041–33050, 2021, doi: 10.1109/ACCESS.2021.3061277.

[16] A. J. Ko and B. A. Myers, “A framework and methodology for studying the causes of software errors in programming systems,” *J Vis Lang Comput*, vol. 16, no. 1–2, pp. 41–84, Feb. 2005, doi: 10.1016/j.jvlc.2004.08.003.

[17] T. Michaeli and R. Romeike, “Improving Debugging Skills in the Classroom,” in *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*, Oct. 2019, pp. 1–7. doi: 10.1145/3361721.3361724.

[18] S. N. Liao *et al.*, “Behaviors of Higher and Lower Performing Students in CS1,” in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, Jul. 2019, pp. 196–202. doi: 10.1145/3304221.3319740.

[19] Mei-Wen Chen, Cheng-Chih Wu, and Yu-Tzu Lin, “Novices’ Debugging Behaviors in VB Programming,” in *2013 Learning and Teaching in Computing and Engineering*, Mar. 2013, pp. 25–30. doi: 10.1109/LaTiCE.2013.38.

[20] M. A. Miljanovic and J. S. Bradbury, “RoboBUG,” in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, Aug. 2017, pp. 93–100. doi: 10.1145/3105726.3106173.

[21] V. C. S. Lee, Y. T. Yu, C. M. Tang, T. L. Wong, and C. K. Poon, “ViDA: A virtual debugging advisor for supporting learning in computer programming courses,” *J Comput Assist Learn*, vol. 34, no. 3, pp. 243–258, Jun. 2018, doi: 10.1111/jcal.12238.

[22] A. Bottcher, V. Thurner, K. Schlierkamp, and D. Zehetmeier, “Debugging students’ debugging process,” in *IEEE Frontiers in Education Conference Proceedings - Frontiers in Education Conference, FIE*, Nov. 2016, vol. 2016-November. doi: 10.1109/FIE.2016.7757447.

[23] T. Michaeli and R. Romeike, “Improving debugging skills in the classroom - The effects of teaching a systematic debugging process,” in *ACM International Conference Proceeding Series*, Oct. 2019. doi: 10.1145/3361721.3361724.

[24] P. Denny, J. Prather, and B. A. Becker, “Error message readability and novice debugging performance,” in *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, Jun. 2020, pp. 480–486. doi: 10.1145/3341525.3387384.

[25] K. Mangaroska, K. Sharma, M. Giannakos, H. Trætteberg, and P. Dillenbourg, “Gaze insights into debugging behavior using learner-centred analysis,” in *ACM International Conference Proceeding Series*, Mar. 2018, pp. 350–359. doi: 10.1145/3170358.3170386.

[26] M. Ahmadzadeh, D. Elliman, and C. Higgins, “An analysis of patterns of debugging among novice computer science students,” in *ACM*, 2005, pp. 84–88.

[27] Y. T. Lin, C. C. Wu, T. Y. Hou, Y. C. Lin, F. Y. Yang, and C. H. Chang, “Tracking students’ cognitive processes during program debugging-An eye-movement approach,” *IEEE Transactions on Education*, vol. 59, no. 3, pp. 175–186, Aug. 2016, doi: 10.1109/TE.2015.2487341.

[28] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, “Debugging: the good, the bad, and the quirky -- a qualitative analysis of novices’ strategies,” *ACM SIGCSE Bulletin*, vol. 40, no. 1, pp. 163–167, Feb. 2008, doi: 10.1145/1352322.1352191.

[29] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander, “Debugging from the student perspective,” *IEEE Transactions on Education*, vol. 53, no. 3, pp. 390–396, Aug. 2010, doi: 10.1109/TE.2009.2025266.

[30] S. Elo and H. Kyngäs, “The qualitative content analysis process,” *J Adv Nurs*, vol. 62, no. 1, pp. 107–115, Apr. 2008, doi: 10.1111/j.1365-2648.2007.04569.x.

[31] R. C. Metzger, “The way of the computer scientist,” in *Debugging by thinking: A multidisciplinary approach*, 2003, pp. 473–507.

[32] A. Cohen and E. Doveh, “More on the Comparison of Intra-Class Correlation Coefficients (ICCs) as Measures of Homogeneity,” pp. 455–459. doi: 10.1016/S1475-9144(05)04020-8.

[33] J. Bennedsen and C. Schulte, “BlueJ Visual Debugger for Learning the Execution of Object-Oriented Programs?,” *ACM Transactions on Computing Education*, vol. 10, no. 2, pp. 1–22, Jun. 2010, doi: 10.1145/1789934.1789938.