Novice Reflections on Debugging

Jacqueline Whalley
Auckland University of Technology
Auckland, New Zealand
jwhalley@aut.ac.nz

Amber Settle
DePaul University
Chicago, IL, USA
asettle@cdm.depaul.edu

Andrew Luxton-Reilly
University of Auckland
Auckland, New Zealand
a.luxton-reilly@auckland.ac.nz

ABSTRACT

Despite decades of literature calling for research on teaching debugging, we still lack clear guidelines on how to teach debugging processes effectively. This is a significant problem as debugging is an activity that is a key component of software development and a frustrating and time-consuming barrier to students who are learning to program. In this paper we reflect on teaching and learning debugging through the lens of student beliefs about their own debugging practice. We find that few students use systematic approaches such as the scientific method of hypothesis generation and testing that is informally adopted by many software developers. Some students appeared to be satisfied with their ad hoc approaches while others expressed a desire to develop more thorough approaches.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

debugging, debugging process, introductory, reflection

ACM Reference Format:
Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice Reflections on Debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432374

1 INTRODUCTION

Despite the importance of debugging in teaching and learning programming, it is an area of the curriculum that has typically been under-emphasized. The observations of researchers in the 1970s, such as "The concepts of debugging are among the most neglected topics of computer science education" [17], and "there seems to be no formal instruction on debugging or guidelines in the text or even any good folklore on how to do it" [28] are equally relevant almost 50 years later. After decades of literature calling for approaches to teach debugging, we still have no clear guidance on what to teach and how to teach it [34].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE '21, March 13–20, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432374

Research has shown that developers tend to debug using a simplified scientific method that consists of cycles of hypothesis generation and hypothesis testing [35] that requires critical thinking. Communication about critical thinking is considered to promote effective learning because it explicitly bridges the key facets of critical thinking: planning, reflecting, and enacting [9, p. 226]. Moreover, communication of critical thinking about "moments of failure" is believed to promote deeper learning and provide insight into how emotion shapes students' selection of debugging strategies [9]. However, educational researchers have tended not to focus on teaching strategies that promote student-driven communication about the critical thinking that surrounds debugging [9].

Brookfield [5] highlighted the importance of reflection in teaching and learning and proposed four lenses for reflection. Our research views debugging through the second lens "the students' eyes" and explores student beliefs about their own debugging practice. Through this lens Brookfield argued that we can begin to identify, evaluate and understand the implications of teaching practices [4]. To this end, we have been investigating the impact of teaching students a structured approach to debugging in closed laboratory sessions in which students are prompted to reflect critically on the steps involved in debugging programs, and collaboratively generate hypotheses for the causes of given faulty programs. This process is aligned with the general scientific method for debugging [35]. However, teaching a systematic process for debugging is only helpful if students adopt the process in their own programming practices. After completing the laboratory activities, students were invited to take part in a study that asked them to debug faulty programs and reflect on the processes they used. In this paper we report students' reflections on their own debugging practices, answering the research questions:

(1) What do students say about their debugging process?
(2) Does reflecting on their debugging process help students to perceive a need for change in their approach?
(3) Do students perceive a structured, formal process for debugging as valuable?

2 RELATED WORK

Based on the constructivist view of new knowledge as being informed by existing understanding, Simon et al. [37] interviewed students to discover the strategies that students applied when troubleshooting. They found that students did not use detailed domain knowledge and did not employ a strategy to understand a system before troubleshooting. Although students bring background knowledge of troubleshooting to computing, they do not approach it systematically.

Several studies have identified the importance of understanding the underlying programming language, and the benefits that prior

programming experience brings to the debugging process [1, 11, 12, 15, 16, 18, 20, 33, 39–41]. Another line of research finds that students who are learning to program have fragile knowledge of programming constructs and lack the experience of experts, making debugging particularly frustrating and time-consuming [2, 3, 10, 29, 32].

In a study of professional software developers, Perscheid et al. [35] found none of their participants were trained in debugging. Programming strategies, including debugging strategies, are not commonly taught in traditional programming courses [8], and there is a notable gap in recent literature on debugging [27], despite decades of calls for further research on teaching debugging [6, 7, 19, 22, 32, 34, 36].

Students who are learning to program can usually fix errors in their code once the location of the bug is known [12], but the process for understanding the code and locating the error is much more difficult [29]. Novices report that finding bugs in their own programs is one of the most difficult issues they encounter while learning programming [24], highlighting the importance of teaching students how to locate bugs in their code.

There are some positive indications that debugging can be taught. In a study of school students learning LOGO, Klahr and Carver [21] demonstrated that students who were taught debugging strategies were able to transfer their knowledge and were more effective at debugging in later programming courses. A combination of paper-based code review exercises and activities that require the correction of faulty source code has been shown to reduce the time needed to subsequently debug programs [7].

Further, it is has been argued that teaching explicit strategies for debugging may help students to develop more effective programming processes [23].

Fitzgerald et al. [12] asked students to complete and then reflect on a debugging task. They found student's perception of their debugging ability was lower than expected, with students demonstrating high capability for finding and correcting faults frequently rating themselves as average in debugging ability. The authors recommend that teachers develop debugging exercises with reflective questions that prompt students to consider the test cases they need and how that information may inform the location of the bug.

Michaeli and Romeike [30] taught high school students a systematic process for debugging based on the scientific method, and found a significant improvement in self-efficacy and debugging performance. In a similar attempt to teach debugging strategies to high school students, Ko et al. [23] observed that although the students saw value in the strategies, many didn't use them. They speculated that several factors, including self-regulation skills, self-efficacy, willingness to delay gratification and prior knowledge, impact on adoption of the taught strategies. These findings reinforce the importance of research involving student perceptions and reflections on debugging, and debugging strategies.

3 CONTEXT OF STUDY

This study was conducted at a mid-sized, urban, research-focused University in New Zealand. Students were recruited from a CS2 course taught in Python. The course requires students to know how to program before entry, but accepts students with familiarity in any programming language to allow transfer from a non-standard pathway. The course therefore provides a brief recap of programming in Python before introducing data structures and algorithms. The course was conducted in the summer semester of 2020. It consisted of six weeks of formal classes including 20 lab exercises each of which was expected to take approximately five hours to complete. One of these lab sessions focused on debugging processes.

In the debugging lab, students were given a brief introduction and a worked example. They were given 20–30 minutes to collaboratively work through a set of paper-based exercises in small groups. Each exercise provided faulty code and asked students think critically and answer the following questions:

(a) **Describe** the difference between the failing test case's actual output and expected output.
(b) What about the failing test case's input **data** is related to the problem?
(c) What part of the algorithm is working incorrectly?
(d) **Describe** what additional information you would need to locate the bug and how you would collect that information.
(e) **Locate the bug** in the code. Which line or lines of code are causing the bug?
(f) **Explain** how you would **fix** the bug.

These questions were designed to prompt students to think about the differences between the problem solution and the program that should produce the solution, as suggested by Gilmore [13]. The process of bug location, which is the focus of the exercises, is aligned with the scientific method described by Zeller [43] and adapted for teaching contexts by Li et al. [25]. After completing the paper-based exercises, students worked through computer-based exercises that asked them to correct faulty code.

4 METHOD

The 103 students enrolled in the course were invited to participate in the study which was conducted in the weeks following the debugging laboratory. None of the research team were instructors for the course. Twelve students, one woman and eleven men, agreed to participate in the study, which consisted of semi-structured interviews conducted immediately following a computer based debugging session. The interviews were transcribed and analysed using a grounded theory approach [14].

4.1 Pre-interview Debugging Session

The debugging session lasted around 45 minutes and consisted of up to four debugging problems. Each of the subsequent four problems had one or more bugs injected and were intended to become progressively more difficult to solve. Not all participants completed all four tasks. Participants were encouraged to move onto a new problem when they appeared stuck, and some participants were stopped after the time for the exercises elapsed but before they had completed the debugging. During the debugging session the process introduced in the debugging lab was only mentioned when students appeared too stuck to make any further progress, as happened with Participants A and B. No other prompts or hints were provided.

Participants were first given a basic off-by-one looping exercise to warm-up, which was completed by all participants. They then

worked their way through the four regular exercises, which were a dictionary-processing function and three complex variants of Soloway's Rainfall problem [38].

The dictionary-processing function and the first two variants of the rainfall problems were a part of the debugging lab for the course from which participants were recruited. The bugs injected into the exercises for the debugging session were different from the ones seen in the debugging lab.

4.2 Interview Protocol

The questions in our semi-structured interview protocol were intended to explore two aspects of student debugging. The first set of questions consisted of reflections on the debugging process that participants had used while trying to solve the computer based debugging problems immediately prior to the interview, namely:

- On problem [x] you seemed to jump directly into trying to solve the bug without examining all the information. Can you tell me why you approached debugging that way?
- Do you feel like you understood how the program was solving problem [x]?
- Could you explain how the program [x] was working?
- You chose to print information in program [x]. How did you decide where to place the print statements?
- How do you feel about debugging?

The final question asked participants to think about whether they might adapt their process(es) in future and in what way they might change their approaches and why, namely:

- Would you change anything about your debugging process after today?

After asking students to reflect on their own debugging, the researchers presented participants with the debugging process discussed in the lab. Participants were asked about the importance of each step in the process.

The interviews were conducted in person and one-on-one between the interviewer and the student programmer. Each interview lasted no more than 15 minutes and were audio recorded and then transcribed prior to analysis.

4.3 Analysis Method

Each of the three authors independently performed an open coding [14], until theoretical saturation was reached after six interviews. The authors then met in order to establish the final set of 40 conceptual labels. Once the set of open codes was established all 12 interviews were classified using those codes. The Fleiss Kappa statistic for three raters showed good agreement beyond chance ($\kappa = 0.645, p < 0.0001$). Any differences in agreement were reconciled by consensus. Finally, axial coding was conducted by all three authors in a collaborative process from which eight core themes related to reflections on debugging emerged. The following discussion is structured by and explores each theme in depth.

5 ANALYSIS

Our analysis identified themes revolving around code comprehension, bug location and information gathering strategies, how difficult it was to locate bugs especially for students who were still learning programming concepts, the emotions evoked during debugging, and the value that students ascribe to methodical debugging processes.

5.1 Debugging Strategies and Approaches

5.1.1 Static Code Comprehension. The process of reading the code provided in the exercises was crucial for two students. Participant C made this general remark about beginning the debugging process: "Well, the first thing I did was when I saw each function within like the set of functions, I'd read the description and see what it does." The longer exercises appeared to be initially overwhelming for some participants, with Participant D reporting: "It's not easy to understand at first glance." They went on to say: "when I looked at the program slightly more, it actually wasn't too difficult to understand", suggesting that a more careful read had helped their comprehension.

Two students specifically discussed the naming of variables. Participant D noted: "well-defined variables actually makes code a lot easier to read" and "the variables have been well-defined enough that you can kind of see what is happening at a given function." Participant L pointed out that variable names chosen by others could make reading code more difficult: "other people use variable names that are different than you do."

5.1.2 Dynamic Code Comprehension. Using print statements to understand small parts of the program was discussed by three participants. For example, Participant D noted: "…the issue for this …was value …it was converting value to an integer…printing right there allowed me to see it wasn't what I wanted".

Printing was particularly important once students began making changes to the code, as noted by Participant E: "If I …make new lists or create …a new dictionary, then I chose to print the new list …and see if it actually appended it or actually popped it or removed it." Participant E added: "When I fix something, I tried to run it or print it …so …I know which output I get and if I'm implementing the right code". In total, six participants used printing to understand the value stored in variables.

Four participants discussed needing to code trace, particularly for the longer rainfall examples. Participant B noted: "And then you can …try to like trace it back from there. Because …you know what you got at a certain point and where you're meant to be at another point". This quote highlights that there may be connections between code tracing and bug location.

Understanding the communication between functions was highlighted by participants, with four participants focusing on function input. Participant E noted: "If I fix [a bug], or if I want to look at which values are getting sent to [a function] like the parameters …then I just print out the argument and I can see if the code is right". Participant F drew connections between function input and possible bugs by noting: "[If] I believe that the input to function is giving me an error, I'll print that input before, so I'll know what I'm actually feeding the functions".

Function output was commented on by six students, with Participant E remarking: "I tend to use the print statement towards the end of a function". Participant H noted: "Basically, whenever I feel like the function is not giving the right output …cause if I read it and it looks like it's fine, but …between this function and the next

function . . . something's gone wrong, then I'll put a print statement in there."

Participant G focused on the type of information communicated between functions, saying this about how to make a choice regarding printing: "Whether the output of a different function is not matching the input type of the next function." Participant C emphasized that checking the function input and output is about understanding, noting: ". . . if I didn't know what data was at a particular instance before it entered or exited a definition, I'd print data to check what it is."

5.1.3 Bug Location. One of the most important aspects of bug location is reducing the search space, this was commented on by six participants. Reducing the search space for the debugging exercises was accomplished in part by using the expected and actual output provided. Participant E noted: "by looking at the test cases I knew . . . which . . . sort of area to fix". Other participants relied on errors for reducing the search space with Participant G saying: "If there's an error seeing where the error occurred that stops the execution of the code that often tells you where . . . the error in the code is".

Reducing the search space can be connected to function input and output, as Participant H makes clear: " . . . there could be lots of functions calling each other and you can't be sure exactly which one is screwing up. So what I do is I put a print function and kind of use them to pinpoint exactly at what point that the data has been, you know, miscalculated or whatever. And that helps me kind of narrow down, okay, it's in this function. "

A similar remark is made by Participant C in reference to the rainfall problem: "I knew that it wasn't in the calculation of the average or anything, it was in the printing parts. So I knew straight away to look at the print_histogram and then from there I realized print_histogram calls back to print."

Being able to speculate on the causes of problems was important for finding bugs in the exercises for four of the students. But the hypotheses were not always precise as this remark from Participant M makes clear: " I had something that I think is wrong and I want to see exactly how it's wrong". Imprecise hypotheses are often connected to the insertion of print statements, as Participant D notes: "[I] put print statements where I think an issue might be." The main method found in the rainfall implementation particularly lent itself to printing to locate problems, as noted by Participant J: "I also put it in the main function. I try to . . . print the data. I know which function is wrong. . . . Which part has problems."

Two participants described their bug-location processes as being somewhat superficial, as exemplified by Participant G: " . . . to see if there were any obvious errors, like really obvious errors, within the code . . . that's why I just read through the code."

5.1.4 Information Gathering. One of the broad strategies identified by seven participants was guessing about bug causes while seeking further information. Multiple students drew connections between guessing and printing, including Participant E: "I kind of like guess [the fix] and check if it's right [with prints]" and Participant F: " I usually have to make a guess about where the error's coming from and then I try to place print statements such that they'll give, give me information." Guessing was described as an intuitive process. For example, Participant L said that they placed print statements "usually following my instinct."

Six participants identified debugging strategies that are dependent on the situation. The placement of prints within loops versus if statements was discussed by Participant M: "Sometimes I do it outside, inside the for loop, depending on exactly how precise I want the information. . . . if I do it inside an if statement . . . if it triggers then there's no point in doing that, which is why I generally put it outside if-else statements." Participant H remarked on code with multiple functions, saying: " . . . with lots of functions calling each other it gets complicated. You need a more methodical approach."

Starting with checking the expected and actual output was mentioned as a broad initial strategy by Participant E: " . . . I kind of compared on the expected output and the actual output."

5.2 Debugging Difficulties

5.2.1 Process Problems. During the reflection four students were critical of the process they used during the debugging exercises. Participant M noted: "I kind of went around in circles." Participant B commented that their process could be better, saying "instead of just trying to wing it and putting random print statements everywhere and making a mess, but yeah."

More complex problems were challenging with respect to bug location, which four participants discussed. In particular, Participant K said: "If I look at the, the very long program, I think that it's pretty difficult for me to look at where it's wrong and . . . how I am going to correct the problem." Participants M and J commented: ". . . if you can't find it . . . you just keep going. Sometimes you can kind of get lost." and "I spent a lot of time in the wrong place."

There was agreement among six participants that debugging someone else's code rather than your own was more challenging because of the need to understand the code. Participant G commented: "If it's my code, I think it's quite easy to debug because I know what's written." Participant G went on to clarify: "Reading other's people's code it's obviously written in a different style. It's a lot more difficult to debug other people's code and that's why I was doing a a lot of reading to try to understand what they were doing." Participant C also agreed that starting with your own code gives an advantage in debugging: "I think it it's debugging my own code it would be a bit easier since if I wrote that code sort of have a general idea where things are, but with . . . other people's code . . . it's a lot harder."

However, two students described how it was difficult to distinguish between what the code was supposed to do and what the code actually did during code comprehension. Participant A described their mental state when viewing other people's code: "You just like tend to go into autopilot and just assume some things are correct even while looking at it." Interestingly, Participant H reported the opposite view: "I can find mistakes in other people's code much more easily than I can . . . find my own mistakes. Um, cause . . . I'm used to reading my code. I know what I expect it to do, but when I read someone else's code, I guess I'm trying to reinterpret it, so really, everything's gonna be as it is rather than what I expect to see I guess and that's kind of like. you don't have a preconception . . . about what it should be doing."

5.2.2 Fragile Knowledge. Some parts of the debugging exercises exposed fragile knowledge in four participants. Certain language constructs were particularly difficult, as noted by Participant E: "I'm not

really strong with dictionaries." Lack of knowledge about built-in Python functions was also an issue. Participant G noted: "I noticed that pop thing earlier, but I thought it was fine because I thought that pop just removed it from the list. But it didn't do that."

5.3 Lessons Learnt, or Not

5.3.1 Attitudinal. Some participant comments expressed attitudes about debugging rather than intentions. Three students indicated that they believe debugging was time-consuming. Participant K noted: "This is quite a lot so I think if I spend more time maybe I can find the bug …"

Debugging also inspired emotional reactions from five students, both positive and negative. Participant L noted: "If you do lots of debugging, you feel you are so tired of it." Participant B said: "I don't mind doing it. …I like having to think about it." Participant C said: "It's a bit tedious …kind of enjoyable …wow, I feel a little accomplished now." Participant F said: "I could avoid it I definitely will avoid it. …it's really annoying. …I'm absolutely not a fan of it. …Yeah, I guess I hate it." And Participant J said: " I quite like it. It's the challenge and [the satisfaction] when I finished the debugging."

5.3.2 Future Intentions. Each participant was asked if they would change anything about their debugging process after the study. Three participants said no, including Participant L: "maybe I'd just use my original style," and Participant F: "to be honest, not really."

Five participants indicated that they would take more care in debugging in the future. Participant M talked about being more thorough, saying: "…just like check absolutely everything," as did Participant H: "I suddenly feel like I need to be more systematic about how I debug." Wanting to be more careful was sometimes a by-product of perceiving their process as being flawed, as illustrated by Participant B: "I probably will follow these steps a bit more, I'd say instead of just trying to wing it and putting random print statements everywhere and making a mess."

Some participants were specific about what they wanted to change in the future. Four students mentioned wanting to read and understand how functions and other smaller pieces of code were used in the larger context of the program. Three students expressed a desire for a more formal process for debugging. For example, Participant K asked: "I just want to ask, is there any …very useful method to debug or just just like this?" Participant C noted: "I'd definitely go back to the debugging lecture and the debugging lab."

5.4 Reflections on Formal Process

Four participants felt that the formal debugging process was either problematic in some way or indicated that they didn't use it. For example, Participant E said: "I just tend to do it my way. Not like not what …what we were taught in the lab." They then added: "[the process could] …be useful in terms of actually solving the questions, but …under …time constraints I think will be kind of hard to, it will take a lot of time to do all the steps." Participant G questioned the utility of specific process steps, saying: "I didn't really need to speculate."

Four participants either indicated that the process was helpful or that they believed their own process included one or more of the steps in the formal process. Participant A praised the formal process, saying: "because it made me look at the problem more linearly rather than …looking at a huge block of functions and not really knowing which one to look at." Participant B noted: "I did find myself looking at the [process] sheet more often. It did give me something to start off and go towards. So yeah, I probably will follow these steps a bit more, I'd say instead of just trying to wing it and putting random print statements everywhere and making a mess." Participant D commented on the utility of the formal process: "[step 1] …very important because if you don't compare the actual output to the expected output you have absolutely no idea what has happened. Actually, I think all these steps are equally important you have to do them to get from one step to another …in order to fix the problem."

Three participants struggled with granularity in debugging, something we dubbed "not seeing the forest for the trees." For example, Participant M noted: "Like with the dictionary thing …I kind of went around in circles. …I feel like I might need to step back sometimes. Try and get the greater picture." And Participant D noted: " …I think probably for a larger program, I think from now on I would probably look at the whole program first …getting a basic understanding of every single function and like a quick understanding of what exactly each function does and how exactly they work or intertwine with each other to solve the problem. So need to see what functions do in relation to the program first before trying to solve the problem."

6 DISCUSSION

6.1 RQ1: Student View of Their Processes

Student discussion of their debugging processes was split between high-level and low-level activities. High-level approaches like reading code, reducing the search space, and forming hypotheses were mentioned, but often in a superficial or imprecise way. Student reflections were dominated by low-level activities such as the location of print statements, identification of variable names or values, code tracing, or examining function parameters or return values. This type of low level focus is also seen in novice programmers [26]. No students mentioned moving between high-level and low-level debugging activities.

When students reflected on their processes some realised that they had largely skipped the code comprehension phase and this has been a fault in their approach. This behaviour was also observed by Yen et al. [42]. Both participants who specifically mentioned reading code described superficial scanning rather than in-depth examination of the functions or documentation. And half the participants specified that debugging someone else's code is more difficult than debugging their own, confirming previous results [24]. The most common reason provided for this perception was that understanding code someone else has written is difficult.

An examination of the three participants who indicated that debugging takes too much time and effort is revealing. None of those participants described themselves as reading code, tracing code, trying to reduce the search space, developing hypotheses, or performing top-down problem location. Two of the three identified guessing as one of the strategies, but one participant did not mention a single debugging approach they had taken. A lack of methodical debugging techniques may be why these students perceive debugging as inefficient.

6.2 RQ2: Student Satisfaction with Their Processes

In some cases, there appeared to be a disconnect between the students' descriptions of their debugging processes, their reflection on the effectiveness of their approaches, and their desire for a change in their debugging processes. For example, Participant L discussed for the most part low-level approaches such as printing variables, identified that they had difficult locating bugs, and possessed fragile knowledge about Python constructs, but did not indicate that they believed their process needed to change in any way. On the opposite extreme, Participant K identified few specific problems with their process, indicating only that they believed it was ineffective, but identified several ways that they wanted to change their approach including taking more care, using more print statements, and learning a formal process for debugging. These results are more mixed than previous studies [12].

For nearly half of the participants, debugging elicited a strong emotional reaction. Roughly half of the emotional reactions to debugging were negative, describing it as tedious, exhausting, and annoying. Among those who described positive emotional reactions, two-thirds discussed the satisfaction that one feels once the debugging is done rather than enjoyment with the process itself.

6.3 RQ3: Perceived Value of a Formal Process

Students lacked methodical debugging processes. They universally characterized their hypothesis generation processes, if present, in vague ways. They described making imprecise guesses and then checking those with print statements, but any sort of systematic approach to code comprehension, bug location, or bug fixes was missing. But only about a third of students explicitly identified their debugging process as being flawed.

Only one student described their process as being ineffective and wanted to learn a more formal process. Although others mentioned specific changes such as needing to take more care or to read and understand the code first. In some cases, the struggle they experienced encouraged students to more seriously consider the utility of a formal debugging process. All of the students who got stuck and were given the process while debugging believed the formal process was useful, whereas only two students who saw the process after they finished debugging viewed it positively.

As noted above, very few students described themselves as having strong code comprehension techniques during debugging, but some of them wanted to change that behavior. A third of the participants identified reading code more carefully as something they wanted to do in the future. A quarter of participants suggested that they wanted to learn a more formal process. And nearly half of the participants wanted to be more careful about their debugging. But there was a distinct lack of overlap between the intention to read code carefully, take more care, and learn a formal process, with only two participants making statements matching more than one intention.

6.4 Implications for Teaching Debugging

Debugging is a frustrating process, and it is therefore understandable that students feel negatively about it. Emotional reactions can hinder learning, confidence and selection of strategies. Our tertiary students' reflections appear to mirror the findings about value propositions from K-12 reported by Ko et al. [23] and highlight the need for instructors to understand and to manage student frustration about and during debugging.

Reflection and thinking critically about learning helps self-regulation and helps achieve deeper learning. Reflection helps students learn from moments of failure and helps to moderate negative emotional reactions. Reflection and meta-cognition can be facilitated through group or pair debugging activities [31] or the use of learning journals [9]. An alternative, more pragmatic, approach might be to integrate steps to elicit critical thinking into a formal and systematic process for debugging.

A formal step-wise process appears to scaffold the processes of bug identification and location by breaking the tasks into manageable chunks. However, students may need to encounter a sufficiently complex debugging task before they appreciate the need for a more systematic process.

Debugging is always likely to be a problem for introductory programming courses because of fragile knowledge, so teaching needs to provide students with more explicit advice about testing code fragments to help with code comprehension. Moreover, any taught debugging strategy may benefit from the inclusion of explicit steps or questions designed to facilitate code comprehension.

7 LIMITATIONS

The students in this study were self-selected and are only a small sample of the students in the class. The class itself is offered in the Summer semester which attracts a subset of the typical student body who are typically more engaged than average students. However, the participants included both students who completed only one or two activities as well as students who completed all activities, which indicates a range of debugging abilities in the study population. Some of the participants were poor at articulating their debugging process, which may have hindered the depth of knowledge gained from the study.

8 CONCLUSIONS

In this study we asked students who had been taught a formal process for debugging to reflect on their practices following an individual debugging session. The participants described using low-level activities, were missing important debugging processes such as code comprehension, and failed to take a methodical approach to debugging. There was not a clear connection between using poor strategies and a desire to change among participants, although the students who used the formal process taught to them when they were stuck believed it to have value. Many students described their feelings about the debugging process as being negative.

It may be that the student reflections are not accurately describing their actual practice in debugging. As such, it would be interesting to compare activities that students are doing while debugging with their reflections on their process. Understanding what students are not describing when reflecting on their debugging may help to illuminate further gaps in their understanding of debugging and aid educators in improving the teaching of debugging processes.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. *SIGCSE Bull.* 37 (2005), 84–88.

[2] Basma S. Alqadi and Jonathan I. Maletic. 2017. An Empirical Study of Debugging Patterns Among Novices Programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 15–20.

[3] Pasquale Ardimento, Mario Luca Bernardi, Marta Cimitile, and Giuseppe De Ruvo. 2019. Reusing Bugged Source Code to Support Novice Programmers in Debugging Tasks. *ACM Trans. Comput. Educ.* 20, 1, Article 2 (Nov. 2019), 24 pages. https://doi.org/10.1145/3355616

[4] Stephen D. Brookfield. 2002. Using the lenses of critically reflective teaching in the community college classroom. *New Directions for Community Colleges* 2002, 118 (2002), 31–38. https://doi.org/10.1002/cc.61 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cc.61

[5] Stephen D. Brookfield. 2017. *Becoming a Critically Reflective Teacher*. Jossey-Bass.

[6] Mei-Wen Chen, Cheng-Chih Wu, and Yu-Tzu Lin. 2013. Novices' Debugging Behaviors in VB Programming. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering (LATICE '13)*. IEEE Computer Society, USA, 25–30. https://doi.org/10.1109/LaTiCE.2013.38

[7] Ryan Chmiel and Michael C. Loui. 2004. Debugging: From Novice to Expert. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA) *(SIGCSE '04)*. Association for Computing Machinery, New York, NY, USA, 17–21. https://doi.org/10.1145/971300.971310

[8] Michael De Raadt. 2008. *Teaching programming strategies explicitly to novice programmers*. Ph.D. Dissertation. University of Southern Queensland.

[9] David DeLiema, Maggie Dahn, Virginia J. Flood, Ana Asuncion, Dor Abrahamson, Noel Enyedy, and Francis Steen. 2020. Debugging as a context for collaborative reflection on problem-solving processes. In *Deeper Learning, Communicative Competence, and Critical Thinking: Innovative, Research-Based Strategies for Development in 21st Century Classrooms*, E. Manolo (Ed.). Routledge, New York, 209–228.

[10] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All Syntax Errors Are Not Equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 75–80. https://doi.org/10.1145/2325296.2325318

[11] Mireille Ducasse and A-M Emde. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering*. IEEE Computer Society Press, Washington, DC, USA, 162–171.

[12] Sue Fitzgerald, Gary Lewandowski, Renée 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.

[13] David J Gilmore. 1991. Models of Debugging. *Acta Psychologica* 78 (1991), 151–172.

[14] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, New York, NY.

[15] John D Gould. 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* 7, 2 (1975), 151–182.

[16] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, Massachusetts, USA) *(CHI '86)*. ACM, New York, NY, USA, 171–174.

[17] R. L. Heilman and G. P. Ashby. 1971. Re-Evaluation of Debugging in the Computer Science Curriculum. *SIGCSE Bull.* 3, 4 (Dec. 1971), 15–18.

[18] David H Jonassen and Woei Hung. 2006. Learning to troubleshoot: a new theory-based design architecture. *Educational Psychology Review* 18, 1 (2006), 77–114.

[19] Yasmin B. Kafai, David DeLiema, Deborah A. Fields, Gary Lewandowski, and Colleen Lewis. 2019. Rethinking Debugging as Productive Failure for CS Education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 169–170.

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

[21] David Klahr and Sharon McCoy Carver. 1988. Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive psychology* 20, 3 (1988), 362–404.

[22] Amy Ko. 2020. The CS-Ed Podcast. https://sites.duke.edu/csedpodcast/2020/01/07/episode-3-amy/ Accessed: 17 March 2020.

[23] Amy Ko, Thomas D. LaToza, Stephen Hull, Ellen A. Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching Explicit Programming Strategies to Adolescents. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 469–475.

[24] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A Study of the Difficulties of Novice Programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Caparica, Portugal) *(ITiCSE '05)*. Association for Computing Machinery, New York, NY, USA, 14–18. https://doi.org/10.1145/1067445.1067453

[25] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a Framework for Teaching Debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '19)*. Association for Computing Machinery, New York, NY, USA, 79–86. https://doi.org/10.1145/3286960.3286970

[26] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITICSE '06)*. Association for Computing Machinery, New York, NY, USA, 118–122. https://doi.org/10.1145/1140124.1140157

[27] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and et al. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106.

[28] Robert F. Mathis. 1974. Teaching Debugging. In *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '74)*. Association for Computing Machinery, New York, NY, USA, 59–63. https://doi.org/10.1145/800183.810443

[29] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[30] Tilman Michaeli and Ralf Romeike. 2019. Improving Debugging Skills in the Classroom: The Effects of Teaching a Systematic Debugging Process. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education* (Glasgow, Scotland, Uk) *(WiPSCE'19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. https://doi.org/10.1145/3361721.3361724

[31] Laurie Murphy, Sue Fitzgerald, Brian Hanks, and Renée McCauley. 2010. Pair Debugging: A Transactive Discourse Analysis. In *Proceedings of the Sixth International Workshop on Computing Education Research* (Aarhus, Denmark) *(ICER '10)*. Association for Computing Machinery, New York, NY, USA, 51–58. https://doi.org/10.1145/1839594.1839604

[32] 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. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. ACM, New York, NY, USA, 163–167.

[33] Paul W Oman, R Cook, Murthi Nanja, et al. 1989. Effects of programming experience in debugging semantic errors. *Journal of Systems and Software* 9, 3 (1989), 197–207.

[34] Devon H. O'Dell. 2017. The Debugging Mindset. *Queue* 15, 1 (Feb. 2017), 71–90. https://doi.org/10.1145/3055301.3068754

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

[36] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article Article 1 (Oct. 2017), 24 pages.

[37] Beth Simon, Dennis Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. 2008. Common sense computing (episode 4): debugging. *Computer Science Education* 18, 2 (2008), 117–133.

[38] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594

[39] Abdallah Tubaishat. 2001. A knowledge base for program debugging. In *ACS/IEEE International Conference on Computer Systems and Applications*. IEEE Computer Society, Los Alamitos, CA, USA, 321–327.

[40] Iris Vessey. 1985. Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.

[41] Leon E. Winslow. 1996. Programming Pedagogy—a Psychological Overview. *SIGCSE Bull.* 28, 3 (Sept. 1996), 17–22. https://doi.org/10.1145/234867.234872

[42] Ching-Zon Yen, Ping-Huang Wu, and Ching-Fang Lin. 2012. Analysis of Experts' and Novices' Thinking Process in Program Debugging. In *Engaging Learners Through Emerging Technologies*, Kam Cheong Li, Fu Lee Wang, Kin Sun Yuen, Simon K. S. Cheung, and Reggie Kwan (Eds.). Springer, Berlin, Heidelberg, 122–134.

[43] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier, San Francisco, CA, USA.