Debugging Techniques in Professional Programming

Amanda Liu 1 and Michael Coblenz 2

1 University of Maryland, College Park, Maryland
2 University of California, San Diego

Abstract

Although debugging is a crucial skill in computer programming, there is no consensus regarding the techniques software engineers use to debug large-scale software systems. This paper describes a preliminary study in which we observed three professional programmers debugging personal and work projects in order to elicit their debugging strategies and techniques. We found that regardless of programming language or task, participants used several common debugging techniques that tended to fall into two modes: identifying the error and fixing the error. We also found that participants rarely used IDE tools but often relied on print statements. Our hope is that once the debugging techniques that professionals use have been identified, the techniques can be integrated into the development of a debugging virtual assistant tool to help aid programmers of all experience levels.

1 Introduction

Although debugging skill is central to success while programming, methods for teaching debugging have been only sparsely studied [1]. One approach, described by Li [1], has been to view debugging as a kind of troubleshooting. Although general-purpose techniques may be helpful, we were interested in identifying specific debugging techniques used by professional programmers. If reusable debugging techniques could be identified, then methods could be developed and added to introductory programming curricula, which in many cases lack dedicated debugging instruction [2].

Although some existing work focuses on techniques used on small codebases, we were particularly interested in techniques that help programmers debug large codebases. These techniques may be more effective at reducing the cognitive burden on programmers: by enabling programmers to make progress using only a partial understanding of the program, they may be helpful for novices, who have difficulty understanding even small programs.

In this paper, we describe a pilot study in which we observed three professional programmers to identify general-purpose debugging strategies used by professionals. Our purpose is to develop a study methodology that we could use on a larger sample of programmers as well as to hypothesize strategies that may be used in debugging. We analyzed the debugging techniques used and compared the contexts in which these strategies were helpful.

We observed eight debugging techniques (table 2), such as tracing through breakpoints and change impact analysis [3]. We found that many of the debugging techniques observed were language and task agnostic. However, different techniques tend to be better-suited toward certain phases. Additionally, despite the availability of debugging tools in their IDEs, our participants rarely used IDE tools, but frequently used print statements to either identify whether a block of code was reached or determine the value of a variable. Our goal is that our research will aid in the future development of a debugging virtual assistant tool that can help novice and expert programmers alike. Our research could also help to inform how computer science curricula teach students debugging skills. Future research can investigate whether the strategies identified in this paper improve debugging success.

Related Work

While some past work has focused on general behavioral patterns during debugging sessions, there is a lack of evidence regarding how professional programmers actually debug. Prior work has described a general hierarchical structure of debugging and compared novice and expert debugging behavior such as the ability to chunk programs [4] [5] [6]. Chattopadhyay et al. conducted a field study about how developers manage context and organize their work while debugging [7] and Beller et al. researched how programmers use external tools such as IDE debuggers to help debug [8]. However, it remains
unclear what concrete debugging techniques professionals use that are reusable across many different contexts. This poses an issue in terms of teaching and improving debugging abilities, as it makes it difficult to provide specific and actionable debugging techniques that programmers can have in their toolkit.

Much of the related work in debugging research has examined the relationship of developers with physical tools, such as IDE-provided debuggers [9] [10] [11]. Beller et al. [8] found that IDE-debuggers are not often used, and many developers primarily use print statements to determine program behavior. These studies have focused primarily on the tool perspective, whereas our study focuses on specific debugging techniques.

Other related work has focused on developer behavior. Jiang et al. [3] observed change impact analysis, which they define as “a general process to identify elements that are indirectly or directly affected by a change,” in debugging behavior in professional developers. Chattopadhyay et al. [7] conducted observations to better understand how developers structure their tasks and manage context. They found patterns in the way programmers organize subgoals of larger tasks, such as sequential, grounding, concurrent, recursive, and alternating. Katz et al. [12] observed students debugging LISP programs and found that debugging behavior differed depending on whether students were looking at their own programs or others’. They found that causal reasoning (finding cause and effect relationships) and backward reasoning (working backward from the end goal) were used more for students’ own programs than others’, and forward reasoning (reasoning from the initial data toward the goal) was used more for other programs their own. Alaboudi et al. [13] observed characteristics of developers’ edit-run cycles (the process of continuously editing and running code while debugging), such as cycle duration and how many cycles were required to fix the error. In contrast with these studies, our work focuses more on specific strategies rather than general behavior patterns.

Several studies have classified developer debugging techniques. Lawrance et al. [14] used an information foraging theory perspective to model the way software developers think about debugging. The study categorized six debugging modes: mapping, drill-down mapping, observe-the-failure, locate-the-fault, fix-the-fault, and verify. In another study, Böhme et al. [15] observed common strategies used by professional developers including forward-reasoning, backward-reasoning, code comprehension, input manipulation, offline analysis, and intuition. Although these studies observed debugging techniques, they focused on broad categories and approaches rather than actionable debugging strategies. Our study classifies actionable debugging techniques and their use contexts, with the hope that this work can contribute to the development of a debugger assistant tool.

Some work has also looked at expert vs. novice developer debugging behavior. Vessey [5] found that experts are more proficient than novices in chunking programs into subproblems and have more smooth-flowing approaches to debugging. Gugerty et al. [6] found that experts debugged more efficiently due to their ability to develop more accurate hypotheses with less study of the code. Our study focuses on debugging techniques used by experienced professional developers, as we hope to develop a tool that can help many levels of developers debug, by using techniques that experts find most helpful.

Certain studies have studied the development of technical tools to aid debugging. Arab et al. [16] contributed HowToo, a platform for programmers to share and request programming strategies and observed that the platform helped programmers to be more systematic and confident in their problem-solving. Similarly, our eventual aim is to use the strategies observed in our study to develop a debugging tool, but ours will be more focused on technique recommendations rather than knowledge-sharing.

3 Methodology
To understand debugging strategies in a real-world context, we conducted a field study in which we observed three professional programmers working on personal debugging tasks. Each observation session was one hour long and followed a think-aloud format, where participants narrated their thought processes as they worked. We were interested in observing debugging techniques used on code with which programmers were already familiar, so we asked participants to select projects they had worked
on recently. While they worked on those projects, we observed their behavior and took note of their debugging techniques. The study was approved by our IRB.

3.1 Procedure
We observed the participants over Zoom while they debugged their projects. Participants shared their screen and followed a think-aloud protocol as they worked. At the beginning of the observation session, we instructed participants to narrate their thought processes and any strategies they implemented. When necessary, we prompted participants with questions such as Why did you make that change? and What are you thinking about now?

3.2 Recruitment
We recruited participants by posting announcements in CS forums, Slack channels, and with snowball sampling. We screened for participants with at least a year of professional experience. Participation was voluntary; we obtained informed consent. For this preliminary study, we recruited three participants.

4 Limitations
As a preliminary study, our findings are based on a limited number of observations from a small pool of only three participants. Due to the small sample size, our study may not be sufficient to draw generalizations about all programmers and programming techniques. Rather, our study is focused specifically on techniques used by professional programmers in their area of expertise.

Participants performed different types of programming tasks and worked in different languages. Rather than generalizability, our study focuses on identifying techniques that appear in various debugging situations, so that these can be further expanded on in future debugging research.

It is challenging to separate debugging work from general coding. Many of the strategies observed can apply to both tasks. Our study simply focuses on the tasks observed in the sessions in the context of debugging.

The observation sessions were only one hour long, limiting our ability to identify participants’ strategies. And, since participants were working on personal projects, they could have spent differing amounts of time familiarizing with the bugs before the debugging sessions. As a result, we may have missed some earlier steps of the debugging process. However, our sessions still provided ample time to observe the participants switching between several different debugging techniques.

Participants’ abilities to complete and understand the debugging assignment depended, to some extent, on the quality of our instructions. To help control for variation, we ensured that all participants received the same instructions at the beginning of the observation session. We asked participants to think aloud to reveal their thought process as they debugged, and we prompted them with open-ended questions throughout the session as needed. Participants also all selected their own personal projects to control for differences in experience and areas of expertise, as each was working on something well-suited to their ability level.

5 Results
We observed three professional programmers working on their tasks. Each programmer brought their own personal project to debug during the one-hour observation session. We took notes as the participants followed a think-aloud procedure, and we synthesize the findings from our notes below.

5.1 Participants
The demographics of the three participants are summarized in table 1. Two participants were industry software engineers, and one was a PhD student in Computer Science. The participants’ years of professional programming experience ranged from 2 to 20+ years. The languages the three participants used for the debugging observation session were JavaScript, Java, and Lisp, respectively. Although Lisp is not commonly used by most software engineers, the Lisp programmer used many of the same
debugging techniques as the other programmers. This supports the idea that debugging techniques are often language-agnostic, and can be applied across many different debugging scenarios.

5.2 Debugging techniques

We used thematic analysis [17] to synthesize results from the debugging observation sessions and identify debugging techniques. Throughout the sessions, we observed several key debugging techniques, which are outlined and described in table 2.

Table 2. Definitions of codes identifying debugging techniques and examples of participant verbalizations.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Reading code</td>
<td>Tracing through code verbally to understand it</td>
<td>“Let’s see what this code is trying to do.”</td>
</tr>
<tr>
<td>(2) Tracing through breakpoints</td>
<td>Setting breakpoints and stepping through code using the IDE debugger</td>
<td>“Now I’ll step into this loop.”</td>
</tr>
<tr>
<td>(3) Change impact analysis</td>
<td>Making changes in code and testing to see if they lead to expected result</td>
<td>“This didn’t do what I wanted.”</td>
</tr>
<tr>
<td>(4) Narrow in on error</td>
<td>Running increasingly smaller blocks of code to determine where the error occurs</td>
<td>“I’m trying to find a minimum example of where this goes wrong.”</td>
</tr>
<tr>
<td>(5) Print statements — tracing execution</td>
<td>Printing arbitrary statements to determine whether block of code is being reached</td>
<td>“It’s not printing, so the code isn’t even reaching this line.”</td>
</tr>
<tr>
<td>(6) Print statements — exposing data</td>
<td>Printing relevant variables to determine data values</td>
<td>“I wonder what this value is.”</td>
</tr>
<tr>
<td>(7) Online searching</td>
<td>Searching or browsing web pages and forums</td>
<td>“This is what StackOverflow says.”</td>
</tr>
<tr>
<td>(8) Requesting help</td>
<td>Asking for help from colleagues</td>
<td>“I’m going to message my advisor and see if he remembers which test case was failing.”</td>
</tr>
</tbody>
</table>

As fig. 1 shows, although each of the participants was working on unique projects in distinct programming languages, many of the debugging strategies observed were present across multiple participants.

6 Discussion

This section discusses four key lessons learned from our experiment. Although our participants had varying backgrounds, programming languages, and tasks, we found that many employed similar debugging techniques. The four lessons we summarize are as follows: (1) Debugging techniques are often language and task agnostic; (2) Debugging tends to occur in phases, and different techniques are better-suited toward certain phases; (3) Print statements can serve different purposes: identifying whether a block of code is reached or determining the value of a variable; (4) Participants rarely use IDE tools.
Table 3. Debugging techniques and tasks for each participant.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Task</th>
<th>Debugging Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Web Development</td>
<td>(1) Reading code, (2) breakpoints, (3) change impact analysis, (5) print statements - execution</td>
</tr>
<tr>
<td>P2</td>
<td>App Development</td>
<td>(1) Reading code, (3) change impact analysis, (4) narrow in on error, (5) print statements - execution, (6) print statements - data, (7) online searching, (8) requesting help</td>
</tr>
<tr>
<td>P3</td>
<td>Function Type Annotator Development</td>
<td>(1) Reading code, (3) change impact analysis, (4) narrow in on error, (6) print statements - data, (7) online searching, (8) requesting help</td>
</tr>
</tbody>
</table>

Figure 1. Debugging Strategy Use Frequency

**Debugging techniques are often language and task agnostic.** Five out of the eight debugging techniques were observed across multiple participants (Figure 1). Two of the debugging techniques, (1) reading code and (3) impact-change analysis, were used by all three participants. Four of the techniques, (4) narrow in on error, (5) print statements - tracing execution, (6) print statements - exposing data, and (7) online searching were used by two participants. Only (2) breakpoints and (8) requesting help were used by just one participant, although all participants had access to a breakpoint debugger. The reason why (2) breakpoints was a less common strategy is further explored at the end of this section. The only participant who used (8) requesting help was the graduate student. A reason for this could be because unlike the two industry professional participants, the graduate student has access to an advisor who is directly knowledgeable about their project, which makes it more useful for them to ask for debugging help. Overall, these findings support the idea that there can be debugging techniques that are applicable across many different programming fields and languages.

**Debugging tends to occur in phases, and different techniques are better-suited toward certain phases.** The debugging techniques identified in this study can be categorized into two main phases: Phase (1) Understanding and/or identifying the error and Phase (2) Attempting to fix the error.

In Phase (1), participants used techniques including (1) reading code, (2) breakpoints, (4) narrow in on error, (5) print statements - line, (6) print statements - variable, (7) online searching, (8) requesting help. Phase (2), or attempting to fix the error, includes the step (3) change-impact analysis. Transitioning from Phase (1) to Phase (2) was not purely linear, as the participants did sometimes return from Phase (2) to Phase (1) after realizing their changes did not have the intended impact. Categorizing debugging techniques can help us understand their purpose and functionality. Lawrance et al. [10] identified observing the failure, locating the fault, fixing the fault, and verifying the fix as four of six key phases in debugging. In our categorization, Phase (1) (understanding and/or identifying the error) encompasses observing the failure and locating the fault, while Phase 2 (attempting to fix the error) encompasses fixing the fault and verifying the fix. The techniques we observed in this study fall into one of these categories. It is likely that the reason we did not observe the other two phases from Lawrance et al. [10] (mapping, drill-down mapping) is due to our study design. Because we only observed the programmers for an hour as they worked on projects they already had substantial experience with, we did not observe them gathering context on the project.
We also note that our contribution in this study lies in identifying the strategies used in these phases, rather than identifying the phases themselves.

Print statements can serve different purposes: identifying whether a block of code is reached or determining the value of a variable. In some cases, participants included print statements without caring about the value being printed, instead focusing on whether the line was run at all. This type of print statement served as a way to understand whether the program was reaching a certain line. In other cases, participants explicitly printed a variable to understand what its data value was. In our study, participants used the first case more than the second. Identifying whether code was reached or determining the value of a variable were just two applications of print statement debugging. Print statement debugging should be further explored since this is such a widely-used technique.

Participants rarely use IDE tools. Similar to findings in related literature, few participants used IDE tools during the debugging session [8] [4]. Only one participant (participant 1) made use of IDE debugging tools during the observation session. This could be for several reasons. For one, many of the debugging techniques used could encompass some functionality of the IDE debuggers (i.e. print line statements and breakpoints). This finding could also support the idea that Computer Science curricula lack a focus on teaching debugging, although this would have to be further assessed through closer study of curricula. We note that the only participant that used the debugger had 20 years of experience, which is significantly more than the other two participants. This could signify a difference in debugging education, between education that emphasized debugging and those that didn’t.

7 Future Work
Future debugging research is needed to produce more generalizable results. While generalizability was not our focus (instead the aim was to explore several debugging techniques), this will be necessary if an effective debugging virtual assistant tool is to be developed. Future studies should recruit more participants, as well as more varied languages and programming tasks. Additionally, while this study only observed participants who received formal college CS education, it could be interesting to analyze differences in debugging techniques between programmers with different educational backgrounds (i.e. bootcamp, self-taught, etc.).

Future work can also further explore the debugging techniques identified in this paper in various contexts, with an emphasis on understanding which debugging techniques are most effective in different situations. One question of interest is whether using and teaching the debugging strategies described in this paper improves debugging success. If a debugging virtual assistant tool is to be developed, future studies can evaluate its utility to help answer this question. Furthermore, our study can be scaled up to identify additional strategies and refine the classifications of the strategies found.

8 Conclusion
The study is the first to our knowledge to identify and classify specific, actionable debugging techniques, which can be applied to developing a virtual debugging assistant. We observed that debugging techniques are often language and task agnostic, which suggests that our findings can be broadly applied to helping programmers improve debugging skills. We also observed that debugging techniques tend to be better suited to either identifying errors or fixing errors; knowing which phase a programmer is in can narrow down the most useful techniques to use. Additionally, we found that participants often used print statements (either to identify whether a block of code was reached or to determine the value of a variable), but they rarely made use of IDE tools. This finding can help inform development of future programming curricula, for example emphasizing formal education of IDE tools. Understanding how professional programmers debug is a first step to helping programmers of all levels improve this crucial skill.
9 Acknowledgements
We would like to thank the participants for their time and contributions to this study. We are also grateful to our PLATEAU mentor, Brittany Johnson-Matthews, for her feedback on our paper.