Quest Journals
Journal of Software Engineering and Simulation
Volume 10 ~ Issue 9 (2024) pp: 09-15
ISSN(Online) :2321-3795 ISSN (Print):2321-3809
www.questjournals.org

Research Paper

Debugging Tactics in Novice Programmers: The Familiarity Factor

Kavyansh Gupta
Kavyanshlu@icloud.com,

***R**eceived 25 Aug., 2024; **R**evised 02 Sep., 2024; **A**ccepted 04 Sep., 2024 © The author(s) 2024.*
P**ublished with open access at **www.questjournas.org

I. INTRODUCTION

Code familiarity refers to a programmer's ability to comprehend a code by recognizing its structure, logic, and components. It can result from previous exposure to similar code patterns, functions, or algorithms. It can also be considered a measure of the programmer's comfort with the code presented to them in terms of code readability. Inspired by the challenges faced by programming students, this study aims to investigate the influence of a Novice programmer's familiarity with the code on the tactics they employed to debug it and their effectiveness. Additionally, This study aspires to draw a comparison between the strategies applied by a novice programmer and those preferred by expert ones. After a conclusion is made, the apex concern will be resolved with the help of a programming project. It was difficult to find previous studies that directly dealt with the concept of code familiarity. Code familiarity is a highly underused term that can be comprehended and used in many different ways. This is probably the main reason why there is not much literature that deals with different levels of code familiarity and how they affect programmers' efficiency while debugging. Extending this line of research could provide valuable insight into how familiarity with code positively and negatively influences a developer's ability to debug code. Also, to extend upon this line of research, currently, there are no studies that investigate how developers maintain and build familiarity over time and how this concept relates to other variables such as team structure and the size and complexity of the codebase. Thus, the current literature surrounding code familiarity suggests that there is a need for more targeted empirical studies aimed at understanding the positive and negative effects of code familiarity on a developer's debugging abilities. This study offers crucial insights into how seeing similar code patterns can boost debugging accuracy even if it doesn't cut down on time. This research has a big impact on our grasp of programming education and tool design showing how important it is to know code patterns well to debug more . The research also points out big differences between how beginners and experts debug. Beginners often look at surface-level stuff and have trouble with complex code, while experts use advanced mental models and test their ideas as they go. These findings stress the need to think about how familiar people are with code and their individual problem-solving styles when creating tools and settings to help both beginners and experts debug .To conduct the research, Ten Introductory (Novice) programming students with similar experience were given a Python code fragment containing five international errors(bugs). They were asked to indicate on a scale of 1 to 5 how familiar they were with the code along with the time taken by them to debug the code, the challenges they encountered, and the tactics applied by them in carrying out the process. Based on their responses, they were divided into groups of varying code familiarity concerning which the rest of the data was analyzed:

II. LITERATURE REVIEW

Challenges Faced by Novice Programmers

Literature emphasizing Novice programmers is wide. However, there are very limited resources available on Code Familiarity among them. The researchers, Lahtinen, Ala Mutka, and Jarvinen reported that novice programmers face numerous challenges while debugging a certain program to solve all the errors including bugs in terms of program construction, such as comprehending programming structures and programming language syntax and dividing functionality into procedures .[1]Meanwhile, Andrew Peterson noted that novice programmers often end up struggling with misconceptions while debugging, difficulty in articulating tactic plans to the computer, lack of explicit understanding, and misunderstanding the functionality of loops and the meaning of variables.[2] Furthermore, Sorva and Vivahienen found that novice programmers are often unfamiliar with debugging strategies and rely on trial and error instead, resulting in frustration and inefficiency . Another major

problem that occurs is correlated with the inability to understand the core purpose of the program while attempting to debug it . [3]On the contrary, Juan D Pinto highlighted that novice programmers may have difficulty identifying certain types of errors, which can hinder their ability to effectively debug their code, particularly in the context of debugging concurrent software . Moreover, the researcher reported that debugging often occurs outside of class, which can be a significant challenge for novice programmers, as they may struggle to identify and fix errors on their own without adequate support and guidance .[4]

Challenges faced by novice programmers in debugging

Helminen suggests that novice programmers face challenges in understanding and using different programming environments, dealing with code fragment problems, and grasping specific programming concepts such as collections, functions, error messages, iteration, outputting results, indentation, and variables . Furthermore, the researcher noted that novice programmers tend to request feedback less frequently when given execution-based feedback compared to line-based feedback, suggesting that students may struggle with understanding the execution of their code and how to debug it effectively. [5]In continuation to previous studies, Juan D Pinto suggested that novice programmers struggle to isolate the problematic code or component causing bugs, fail to form robust mental models of the problem and solution, and have difficulty explaining the purpose of code and understanding its behavior in context . Additionally, the researcher found that novice programmers may overlook security vulnerabilities in their code, as their priority is producing error-free code rather than ensuring security .[6]

Tactics Employed by Novice Programmers

In prolonging his analysis, Lahtinen reported that novice programmers employ varying tactics to debug their code, for instance, adding a statement to print the contents of a variable and code tracing, which allows the programmer to comprehend the step-by-step execution of the code.The researcher also observed that novice programmers use pattern matching, where they change the code that "didn't look right", and isolation of code to discover the problematic area through commenting out or altering the code.[1] Inversely, Felix Lee argued that novice programmers employ tactics dealing with a dry run. For Instance, testing their code, taking notes on paper to process information more effectively, and asking people in their social networks for help . The researcher also concluded that novice programmers consider their past experiences using particular debugging tactics and their experience in their debugging sessions when deciding which debugging tactic to use .[7]

Pasquale Ardimento and team on prolongation highlighted the effectiveness of various debugging strategies, including the reuse of known bugs, hypothesis generation, and strategic navigation . The researchers inferred that these strategies aid novice programmers in identifying and correcting errors in their code, and also help them to develop problem-solving skills and critical thinking. Additionally, the researcher reported that novice programmers can benefit from error detection and correction, marking, and annotation, and strategic question categorization ultimately succeeding in the debugging process.[8]The researcher, Manuel A. observed that tactics such as reading code, tracing execution, debugging, and deconstruction techniques can also aid novice programmers in identifying and correcting errors in their code.[9]For example, a novice programmer may use code tracing to understand the execution of their code, as shown in the following Python code:

```
def add_one(number): return number + 1
def multiply_by_three(number): return number * 3
print(add_one(multiply_by_three(5)))
```

III. METHODOLOGY:

The methodology employed in this research paper involved the recruitment of approximately 10 students with homogeneous levels of programming experience to participate in this study through a pre- experimental questionnaire that collected data related to the participants' experience with Python programming, willingness to participate in the study, basic concepts of python and hands-on ability in running computer applications. Therefore It was assured that all the participants had all the necessary prerequisite knowledge of Python programming and computer applications to contribute to carrying out the procedure smoothly. A Python code containing 5 intentional bugs along with an algorithm of the code was created based on the concepts of Python and the logical ability of the participants to comprehend the purpose of the code with the help of the provided algorithm and its influence on the given code and bugs in it. Before starting the procedure, a quick and crisp overview of the study's methodology was given to participants. However, the main idea of this research was kept un-told since the participants may adjust their natural course of debugging the code to improve their performance which will hinder the final result. Throughout the experimental procedure, meticulous records were kept of the participants' debugging processes, including the time taken, strategies employed, and familiarity Following the completion of the debugging tasks, participants were invited to provide feedback through a post-experiment questionnaire, assessing their perceived difficulty, confidence levels, and overall satisfaction with the debugging process. The

data was analyzed in-depth by both descriptive and inferential statistical analyses to compare if the time spent, strategies employed, and challenges faced, across different levels of code familiarity, showed any difference in debugging efficiency as identified by the number of bugs accurately resolved. Considering these factors, the participants were divided into two groups, Group A which comprised less adept novice programmers who had a familiarity of 1 to 3 on a 5-point scale, and Group B which comprised slightly more adept programmers who had a familiarity of 3.5 to 5 on a 5-point scale. for more in-depth analysis, participants having identical familiarity with the code were kept in the same sub-group. Now, the average of the time taken and the number of bugs resolved by each group were calculated using the following formulas :
Average time taken =sum of the time taken by the participants falling in that group number of participants falling in that group
Average no. of bugs resolved =sum of bugs resolved by all participants falling in that group number of participants falling in the group
This data was later used to compare strategies employed by novice programmers with varying levels of familiarity with the code.

Given Python code :

```
While True:
Age = int(input('Enter your age:'))
Break
Except ValueError:
print('Invalid Input. Please enter a valid number)
If age >= 18:
print("You are eligible to vote")
else
print("You are not eligible to vote")
print("Counting down from your age:) for i in range(age , -1, -1):
print(i, end=" ") print()
Count = 2
While count < age: Count == 1
print("Are we there yet?")
If age >= 21:
print("You are legally allowed to consume alcohol")
else:
print("You are legally not allowed to consume alcohol")
for i in range(1 , age + 1): If age % i == 0
print(i, "is a factor of your age")
Input("End of Program")
```

Provided Algorithm :
Check if age is greater than or equal to 21 and print eligibility for alcohol consumption Check if the age is greater than or equal to 18 and print eligibility to vote.
Display the factor of the user's age using a loop
Print an extra statement at the end of the program.

Intentional bugs :
1.Line 8: A missing colon after "else" (Structural syntax error)
2.Line 10: A missing quotation mark after the print statement, before the bracket (structural syntax error) 3.Line 14 : 0 instead of 2 (Algorithm interpretation and critical thinking)
4.Line 16 : '>=" to come instead of "==" (conceptual and critical thinking")
5.Final line: Print to be used instead of Input.

Rectified code :

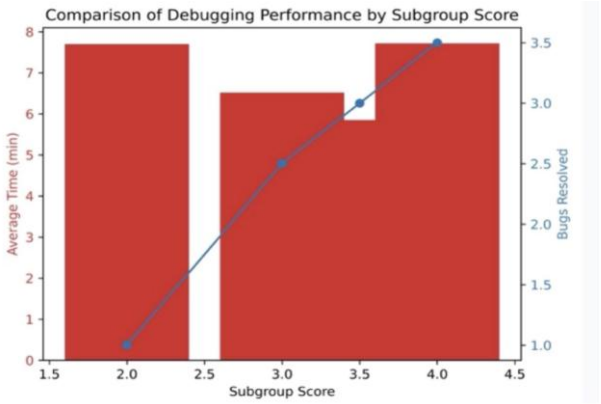
```
While True:
Age = int(input('Enter your age:'))
Break
Except ValueError:
print('Invalid Input. Please enter a valid number)
If age >= 18:
print("You are eligible to vote")
```

```
else:
print("You are not eligible to vote")
print("Counting down from your age:") for i in range(age , -1, -1):
print(i, end=" ") print()
Count = 0
While count < age: Count >= 1
print("Are we there yet?")
If age >= 21:
print("You are legally allowed to consume alcohol")
else:
print("You are legally not allowed to consume alcohol")
for i in range(1 , age + 1): If age % i == 0
print(i, "is a factor of your age") print("End of Program")
```

IV. RESULTS:

Number of bugs resolved and time taken

After gathering all the information the participants were split into two groups depending on how they knew the code rated on a scale from 1 to 5. Group A was made up of beginner programmers with familiarity scores ranging from 1 to 3 while Group B consisted of skilled programmers with scores between 3.5 and 5. To delve deeper into the data participants with familiarity levels were sorted into subcategories. In Group A there were subcategories for those with scores of 2 and 3. The subgroup with a score of 2 included three participants who took an average of 7 minutes and 42 seconds to fix issues resolving one out of five bugs on average. The subgroup with a score of 3 had four participants who took an average of 6 minutes and 31 seconds to debug fixing around two and a half bugs out of five. In Group B participants were further categorized based on scores of either 3.5 or 4. The subgroup with a score of 3.5 had one participant who spent an average of five minutes and fifty-one seconds debugging successfully resolving three, out of five bugs. In the subgroup where both individuals had a familiarity score of 4 two participants spent an average of 7 minutes and 43 seconds debugging successfully addressing 3.5 out of 5 bugs. This data has been depicted in figure 1: Comparison of debugging performance by sub-score(code familiarity).



This graph shows the average time taken (in minutes) and the number of bugs resolved for each subgroup score. The red bars represent the average time taken to debug, while the blue line with markers indicates the number of bugs resolved.

Challenges encountered while debugging

For Group A, which were less adept novice programmers and had a familiarity score between 1 and 3, the main issues were difficulties in understanding the programming structures and syntax. These participants struggled with knowing the primary purpose of the program they were trying to debug, so they made faulty assumptions and applied inefficient debugging strategies. in addition, It was said that participants from this group of novice programmers had common misconceptions about the role of some minor aspects, like not being able to decide the importance of accuracy of the value of a specific variable that is involved in some calculations, and the greater than, equal to and lesser than functions in the for loop and while loop functions in achieving the purpose of the code. in addition, the participants were perplexed due to the complex code structure and found the algorithm to be a little less informative in terms of the purpose of specific steps. the participants reported that the code was a little difficult in terms of length, which ultimately led to anxiety among them thus hindering their debugging

performance. surprisingly, Participants in the group pretty much faced the same challenges as faced by participants of group A; the only difference between them was that they were able to analyze which possible errors could be made in a specific line of code due to a higher rate of familiarity, leading to an increased number of bugs resolved. Yet, they also encountered challenges like anxiety, which was slightly less but still influential due to the vast length of the code despite higher familiarity with it. It seems, however, that they could cope with the hindrance of complex code structure by being familiar with it.

Tactics and strategies used to debug the code

Participants from Group A took an all-round approach to debugging the code. They first started with a grand strategy of reading the code. They skimmed through the whole codebase, taking heed of details, and tried to trace any blinding errors or blemishes. Moreover, they got back to the old-school technique of pen and paper to take a red run at how the code would execute. This allowed them to simulate the program's flow step by step, gaining an in-depth understanding of its behavior and where potential pitfalls may occur. Members of Group A were further determined to try and relate the code to some basic Python concepts, which ensured that the code followed the syntax and semantics of the Python language. This certainly helped to shorten the time for detecting and fixing errors since they could find areas in which they differed from the expected norms by relating the code to the fundamental principles. In addition to this, they compared the problematic code with known working examples. This comparison would let them identify discrepancies and inconsistencies that guided their debugging in a much more effective way. In addition to this, Group A members were always keen on seeking opinions and feedback from their peers, which allowed ideas from others to easily be considered in the debugging process. Here, they capitalized on the collective competence and experience of their colleagues to find some hidden issues and explore alternative solutions. Finally, they remained watchful for common syntax errors, such as indentation problems and faults of a typographical nature, which are an eyesore to most. By keenly looking at these common pitfalls, they managed to correct them quickly, thus retaining the code's integrity.

In contrast, Group B participants were methodical and broke the code into chunks they felt were digestible and would not overwhelm them while analyzing it. This strategic move helped them pay close attention to the sections of the code, which thus enabled closer inspection to be performed concerning the logic and function of the code. They checked every piece of the code against the algorithm provided, taking meticulous care to check if there was any deviation or disparity within the algorithms that could potentially form a fault. Additionally, members of Group B used the strategic addition of print statements to check the contents of variables and the flow of execution. By strategically placing such diagnostic tools at vantage points within the code, they could get valuable knowledge of its operation as well as possibly obtain the point of failure. They also applied the concept of isolation, where they commented out lines of the code to observe the behavior of the program. Such an isolated approach helped in reducing the problematic area, therefore narrowing down the root cause of the problem more effectively. Group B also took the help of experiences from previous encounters with such codes and debugging sessions and used the knowledge to strategize their approach to debugging. With previous encounters to draw from, they can base decisions on a plethora of knowledge, using tactics most effective in the face of the unique problems that arise. Conclusion Participants from both Groups A and B demonstrated impressive diligence and resourcefulness with their strategies for debugging and managed to combine a blend of analytical skills, collective effort, and strategic thinking toward uncovering and resolving issues

V. DISCUSSION:

Interpretation and analysis of results

In this section, It was attempted to compare the debugging tactics of the novices as in this study and, to some extent, those of the experts given their familiarity with the code, concerning established results of past studies. The analysis involved correlating the same tactics against, first, the challenges that they encountered in debugging the code, and second, the amount of time spent on debugging the code and how effective the tactic was, as measured by the number of bugs resolved. The debugging tactics novices employed in our study—such as code tracing and printing variable contents—were also followed by earlier research, which concluded that novices relied on surface-level features of the code and had difficulties in understanding programming structures and syntax. Experts in this study, however, employed more sophisticated methods of binary search, mental models, and hypothesis-driven debugging, which is also per established results: experts build complex mental models of the software systems and use these mental models to predict where faults are likely to be located ((Lahtinen, E., K. Ala-Mutka, H. M. Järvinen. A Study of the Difficulties of Novice Programmers. 2005, pp. 15-20.)) The correlation of code familiarity with debugging effectiveness is also accorded in prior research, which suggests that bug localization and fixing success are positively correlated with code familiarity. The challenges that novices face during bug fixing, such as "difficulties in understanding programming structures and syntax", are also consistent with the dilemmas committed by novices on features of programming, which divide functionality into procedures ((Sorva, J., A. Vihavainen. A Think-Aloud Study of Novice Debugging. pp. 1-8.)).

Moreover, the observation that higher-familiarity experts still invested a similar amount of time in debugging their programs as lower-familiarity novices suggests that while higher code familiarity may indeed lead to more effective debugging, it may not necessarily result in faster debugging times, corroborating the existing view suggesting that experts may be more careful and thorough in their debugging approach .In this respect, the results of this study support the idea that debugging represents a complex cognitive activity that interacts with several factors among them: code familiarity, debugging experience, and individual differences in strategies for problem-solving ((Pinto, J. D. Investigating Challenges of Debugging Tasks in an Undergraduate Introductory Programming Course)). The improved performance of higher-scorers within the novice population {i.e., score 3}suggests that even small to moderate amounts of code familiarity could be of considerable advantage for debugging results, reconfirming previous findings that code familiarity can be augmented through practice and experience ((Pinto, J. D. Investigating Challenges of Debugging Tasks in an Undergraduate Introductory Programming Course)). Again, this finding demonstrates the challenge taken into account while designing debugging tools and environments that support effective debugging practices in this complex interplay between code familiarity, debugging experience, and individual differences in problem-solving strategies.

The results of this study also indicate that the relationship between code familiarity and effectiveness at debugging is more complex and multi-faceted. Although higher code familiarity is associated with better debugging performance, there may be other influencing factors on the debugging process. For example, code complexity, the nature of the bugs, and the experience and skills of an individual in debugging of the code may all have noticeable effects on the strategies for good results. Moreover, the findings of the study suggest that debugging could be done using different tactics and strategies, where experts use the mental models of prediction for the most preferable places for the bugs to be located. This is confirmed by research that states that experts build complex mental models of software systems and use them to direct their debugging efforts .[3]

It can be inferred from this study that, in general, the role code familiarity plays in debugging effectiveness has been confirmed, and that both novices and experts use different tactics and strategies during the debugging task, which are conditioned by different degrees of code familiarity. More importantly, the study found that such consideration of complex interactions between code familiarity and debugging experience with individual differences in problem-solving orientations are necessary prerequisites when designing tools and debugging environments that can foster effective debugging among both novices and expert programmers. The contribution of the results of this study is therefore the enhancement of further research in the area of involved cognitive processes, which may indeed lead to the development of both more advanced debugging tools and environments that can effectively support novice and expert programmers.

VI. CONCLUSION:

After critical analysis and interpretation of the results, It can be concluded from this study that, with repeated exposure to similar code patterns, functions, or algorithms, one might be able to increase their accuracy in the process of debugging a code but it does not guarantee a reduction in the amount of time taken to do so. Also,the vast difference between the tactics applied by novice programmers and those applied by expert programmers in this process was assessed . It was found that novices relied on surface- level features of the code and had difficulties in understanding programming structures and syntax. Experts in this study, however, employed more sophisticated methods of binary search, mental models, and hypothesis-driven debugging, which is also per established results: experts build complex mental models of the software systems and use these mental models to predict where faults are likely to be located. Also, Novice programmers who were more adept dealt with the debugging process with a technical approach but the ones who were less adept relied on basic methodologies dealing with core concepts and dry-runs. Also, more adept programmers relied on their past experiences while choosing a debugging strategy. More importantly, the study found that such consideration of complex interactions between code familiarity and debugging experience with individual differences in problem-solving orientations are necessary prerequisites when designing tools and debugging environments that can foster effective debugging among both novices and expert programmers. The primary challenges faced by novice programmers were found to be that they misinterpret some minor aspects of code, struggle with complex code structures, and find algorithms perplexing, leading to anxiety that hinders their performance. However, with higher familiarity, novice programmers were able to analyze potential errors more efficiently and therefore resolve more bugs, although they still encounter anxiety due to vast code length which can be overcome eventually by repeated exposure to similar codes.

REFERENCES:

[1]. Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. "A Study of the Difficulties of Novice Programmers." 2005, pp. 15-20.
[2]. Petersen, Andrew, Michelle Craig, and Daniel Zingaro. "Software Debugging Patterns for Novice Programmers." pp. 1-16.
[3]. Sorva, Juha, and Arto Vihavainen. "A Think-Aloud Study of Novice Debugging." pp. 1-8.
[4]. Pinto, Juan D. "Investigating Challenges of Debugging Tasks in an Undergraduate Introductory Programming Course."

[5]. Helminen, Juha, Petri Ihantola, Ville Karavirta, and Lauri Malmi. "How Do Students Solve Parsons Programming Problems? — An Analysis of Interaction Traces."
[6]. Pinto, Juan D., et al. "Investigating the Relationship Between Programming Experience and Debugging Behaviors in an Introductory Computer Science Course."
[7]. Lee, Felix, and James A. Jones. "Exploring How Novice Programmers Pick Debugging Tactics When Debugging: A Student's Perspective." eScholarship, pp. 4.
[8]. Ardimento, Pasquale, Mario Luca Bernardi, Marta Cimitile, and Giuseppe De Ruvo. "Reusing Bugged Source Code to Support Novice Programmers in Debugging Tasks." ERIC, pp. 10.
[9]. Pérez-Quiñones, Manuel A., Kris Jordan, and Colin R. Tucker. "Learning by Taking Apart: Deconstructing Code by Reading, Tracing, and Debugging." Academia, pp. 15.
[10]. Corritore, C. L., and S. Wiedenbeck. "What Do Novices Learn During Program Comprehension?" 1991, pp. 256- 258.
[11]. Pennington, N. "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs." 1987, pp. 132-135.
[12]. Javidmanesh, Mohammad, et al. "How Does Code Familiarity Impact Bug Localization in Programming?" 2013, pp. 4.
[13]. Ehrlich, Susan W., et al. "Expertise Remakes Debugging Strategies." 2008, pp. 412.
[14]. Khan, Sanaullah, et al. "The Effects of Familiarity and Code Complexity on Programmers' Bug Fixing Performance." 2013.
[15]. Javidmanesh, Mohammad, et al. "How Familiarity with Code Affects Bug Localization During Debugging." 2012.
[16]. Ko, A., and B. Myers. "Finding Causes of Program Output with the Java Whyline." ACM SIGCHI Conference on Human Factors in Computing Systems, 2009.
[17]. Vessey, I. "Expertise in Debugging Computer Programs: A Process Analysis." International Journal of Man- Machine Studies, vol. 23, no. 5, 1985, pp. 459-494.
[18]. Lawrence, J., M. Burnett, and B. Dorn. "To See or Not to See: The Effectiveness of Code Highlighting for Rapid Defect Detection." IEEE Transactions on Software Engineering, 2013.
[19]. Fritz, T., G. Murphy, and E. Hill. "Does a Programmer's Activity Indicate Knowledge of Code?" ACM SIGSOFT Software Engineering Notes, 2014.
[20]. Storey, M.-A., T. Fritz, and J. Aranda. "How Do Software Developers Use Code Annotations? A Case Study." IEEE Transactions on Software Engineering, 2014.
[21]. Sillito, J., G. Murphy, and K. De Volder. "Questions Programmers Ask During Software Evolution Tasks." ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2006.
[22]. Zeller, A. Why Programs Fail: A Guide to Systematic Debugging. Elsevier, 2009.
[23]. Barr, E., M. Harman, and P. McMinn. "The Oracle Problem in Software Testing: A Survey." IEEE Transactions on Software Engineering, 2014.
[24]. Weiser, M. "Program Slicing." IEEE Transactions on Software Engineering, 1984.