Past and Present Methodologies for Instructional Debugging in the Age of Generative AI

Connor Magnuson

Abstract—**Debugging is often underemphasized or omitted in computer science curricula despite being a critical skill for success in the software development industry. This paper surveys five key studies regarding the process of debugging, how the skill has been taught throughout time, and how generative AI serves as a platform to advance its teaching. Through this analysis, a comprehensive perspective on the reasons why the skill is difficult to teach is developed. It is stated that careful guardrails and foresight must accompany this new technology to ensure that increased independence rather than over reliance is the outcome of its application to education. Finally, future research is proposed and the increasing importance of debugging skills in a world infiltrated by AI-generated code is discussed.**

Keywords—debugging, teaching, generative AI, LLM

I. Introduction

Software developers must be proficient in two distinct skills — programming and debugging [1]. Though these skills are often considered indistinguishable, they require distinct mental models and knowledge bases and are unequally emphasized in computer science curricula [1]. The latter is often marginalized or entirely omitted from formal instruction despite its vitality for industry success [2]. This can be largely attributed to collective instructor sentiment that teaching debugging is challenging, overly time-consuming, or unscalable [1].

Despite this, the literature has conducted studies examining the teaching of debugging and the viability of doing so throughout time. Unsurprisingly, as software development methodologies and instructional technologies have changed, likewise have the debugging techniques used [1]. As expected, some of the greatest changes were observed when students gained access to computers and independent development tools [1-2]. This paper argues that another monumental shift in debugging strategies is imminent due to the advent of generative AI.

Over the last 5 years, the explosion of generative AI and large language models (LLMs) has triggered a paradigm shift in software development practices and debugging strategies for professional software engineers, computer science students, and computer science instructors alike [3-4]. Whereas some view and utilize these tools as a detriment to problem solving and mental model development, they can also be used as a tool to strengthen debugging skills [3]. From generating debugging exercises with human-like mistakes to explaining test coverage, current research is investigating the use of LLMs to prepare the next generation of debuggers for a world in which generative AI aids in generating code at an unprecedented pace [4].

Through a comprehensive analysis of debugging and its teachability throughout time, this paper develops a comprehensive perspective on the reasons why the skill has been difficult to teach in the past. In addition, the use of generative AI to advance instructional debugging is examined and guardrails are proposed to promote positive outcomes from the application of this technology to the field. Finally, this paper suggests future research topics and the increasing importance of debugging skills given the quantity and characteristics of AI-generated code.

II. Summary

An Analysis of Debugging and its Independence from Programming

In order to understand why debugging is difficult to teach, the skill must first be precisely broken down and analyzed. Several studies have not only analyzed debugging, but also differentiated debugging techniques in novices and experts.

Many with longstanding software development experience or employment fail to decouple the skills of programming and debugging, instead referring to them collectively as the task of software development or even "programming." However, these skills are entirely independent — whereas programming is creative, debugging is reactive [1]. Though both are integral parts of the day-to-day tasks of a software developer, these two skills require entirely separate mental models and strategies [1].

In addition to their importance for industry success, studies have shown that possessing mature debugging skills has significant non-cognitive importance. Researchers note the fine line between beneficial and harmful persistence and how knowing this line can aid in preventing frustration and increase self-efficacy [2]. Whereas some persistence is inherently necessary to troubleshoot the unknown source of a bug or malfunction of a program, continuously fidgeting with a piece of code with no forward progress can be unproductive and discouraging, especially for novices [1-2].

One of the factors contributing to the difficulty of teaching debugging is its complexity. In addition to being heavily dependent on project context, most researchers agree that the task has three phases — creating bugs, finding bugs, and fixing bugs [1]. As a result, studies have targeted one or more of these phases at different times. The ability to find bugs and the ability to then resolve them are not necessarily correlated skills and may develop unequally in different individuals [1, 5].

Furthermore, targeting bug creation is inherently complex given that software errors can be expanded into three distinct subtypes [2]. Syntax bugs, which manifest at compile-time (or even when a program is linted in some modern IDEs) are expressed as language-dependent error messages of varying interpretability [1]. Semantic bugs manifest at run-time, often resulting in cryptic error messages or unexpected crashes which require external tools to analyze [2]. Finally, logical bugs, the most broad and difficult to debug subtype, do not cause a program to fail compilation or execution but instead result in unexpected output [1].

One study attributes these distinct practical bug manifestations to a single abstract "superbug," or gaps in the mental approaches of novice debuggers [1]. Arguing that most novice issues are systemic and transcend language or environment-specific constructs, [1] notes that many novices created bugs as a result of their flawed expectation that a computer or compiler "understands" human intentions regardless of implementation issues [1].

Some studies have also sought to compare debugging strategies in novices and experts, noting significant differences. One such study employed a unique visual interface only allowing users to reveal a small portion of the program they were debugging at a time. The results of this experiment accentuated differences in the breadth of searches employed by the two groups, with experts being more likely to analyze the broader program and the buggy code snippet in context [1]. Additionally, studies find that one reason expert debuggers have an extra advantage is that they can apply prior debugging experience and heuristic/pattern-matching techniques [1]. Some studies even found that whereas experts often addressed the root cause of the issue without changing unnecessary portions of the program, novices were more likely to introduce additional bugs and modify working code when debugging [5].

In summary, this analysis demonstrates the difference between debugging and programming, meriting their consideration and instruction as distinct skills. While programming is a creative task often influenced by language and environment constructs, debugging is a universal abstract reasoning skill essential for all developers. This distinction, combined with its deep connection to individual thought processes, lays the groundwork for its instructional difficulty.

Previous Methodologies of Teaching Debugging and Their Limitations

Given a comprehensive analysis of debugging as a skill as well as the types of issues debugging seeks to resolve, previous studies concerned with the teaching of debugging can now be examined to analyze its teaching complexity and the ways in which instructors have attempted to circumvent instructional barriers.

Various studies have explored changes in debugging performance when novices are given feedback in various forms provided at different times throughout the compilation and execution process. Various media for such feedback include but are not limited to (un)enhanced compiler warnings, instructor comments, and personal teaching assistant support [1]. These feedback forms have been varied both independent of and dependent upon bug type, revealing that personalized feedback contributes most directly to increased learning [2]. However, instructors cannot practically provide personalized feedback in a large course setting, leading to the absence of debugging in curricula.

When studies have changed feedback timing independent of bug type (ex. providing compile-time errors for semantic bugs), one study revealed that "while immediate feedback improves short-term performance, delayed feedback is more effective for long-term conceptual retention" [2]. This does not take into account, however, the non-cognitive tradeoffs of these approaches, which are likely to favor sooner feedback for novices.

Several studies have shown that the ability to converse with others (even of the same skill level) while debugging can dramatically increase performance [2]. Studies attribute this to benefits of cognitive load-sharing and the deeper comprehension gained through verbal discussion [2]. This suggests that although instructors are incapable of providing personalized feedback to all students, assembling students into small groups when learning debugging may allow students to reap some foregone benefits. There are also downsides to this approach, however, such as matching students with different debugging skill levels. This may lead to one student being unsympathetic of other students' concerns or confusions, or may silence otherwise beneficial questions from less-experienced students.

One often underemphasized debugging aid is the creation of unit testing suites. Studies have found that when given the option, novices do not prioritize test creation, highlighting a potential lack of emphasis in curricula [5]. When provided some test cases in a debugging environment, students were extremely unlikely to create additional test cases [2, 5]. This is not only concerning for the purpose of developing skills, but also given that developing test suites is often a task of developers in industry and something that degree graduates should be capable of.

Discouragingly, almost all such traditional studies found that students were unlikely to directly apply the debugging skills they developed through specifically targeted exercises to future coursework [1]. However, studies found that requiring students to reflect on their mistakes and strategies or abstract them away from code improved this metric and encouraged retention [2].

The complex and/or personalized methodologies which cultivated improved novice debugging performance in these studies highlight the major practical tradeoff between quality and scalability that has served as an inhibitor for the skill's instruction over time. However, generative AI now serves as a tool which can achieve both personalized, quality debugging instruction and practical scalability.

Debugging in the Age of Generative AI

While some view Large Language Models as detrimental to the software development industry, they can instead be used as a tool to teach debugging like never before and prepare developers to debug AI-generated software.

It has been shown that the long-term benefit or detriment to users of AI in the context of programming is largely determined by the type of prompting employed [3]. Lazy prompts such as "fix this code" or "make it work" followed by a large code snippet serve to offload cognitive work and critical thinking, thereby eradicating the critical problem-solving experiences necessary for a developer to become a good debugger [3-4]. On the other hand, productive prompts such as weighing design tradeoffs or analyzing vulnerabilities for a given section of code can help developers expand their knowledge base and debugging skills at an unprecedented pace [3-4].

Assembled carefully, large language models can act as chatbot pair programmers of equivalent caliber to the prompter, bringing about all of the benefits of pair programming discussed in the prior section [3]. Additionally, chatbots can be trained to respond in specific ways and within specific limits set by course staff, providing similar services to a TA or professor [3]. One such chatbot developed for Harvard's CS50 introductory programming course was made available to students for assignments and general course questions [3].

Perhaps one of the most intriguing and promising uses of generative AI to enhance debugging skills is its employment in the generation of code snippets with human-like bugs, a natural feature of large language models [4]. One such study created a tool called HypoCompass wherein AI models were used to generate buggy code snippets and novices played the role of an instructor or course assistant, helping the AI "students" debug their issues and justifying the reasoning for their decisions [4]. Students reinforce concepts and heuristics that improve debugging skills and speed when they must teach someone (or something) else their methodologies [4].

Rather than simply wrapping a popular LLM-based chatbot with prompts to generate code, the HypoCompass study used far more complex architecture and provides a glimpse into the kind of thinking required when employing generative AI in the field of instructional debugging. After proposed code samples were generated, they were filtered and evaluated by both AI and humans before being presented to participants [4]. Users also had to explain their reasoning to the "student" chatbot when attempting to debug, and careful guardrails were imparted upon this chatbot to ensure it asked the right type of clarifying questions and did not give solutions to the user [4]. Researchers analyzed not only the success of students in proposing correct bug fixes, but also explanations and reasoning [4]. This study provides one such example of how generative AI can be employed to improve the effectiveness and feasibility of instructional debugging.

Another promising use of generative AI models is aiding in the creation of increasingly comprehensive test suites. Models are not only good at determining untested vulnerabilities and generating corresponding test coverage, but also analyzing patterns in failed test cases and backtracking to problems in the source code [3-4]. When integrated into novice tools, this can reduce the burden of creating test cases, thereby aiding students in debugging and reducing frustration.

III. Conclusions

Though debugging is a skill critical for success in the software development industry, computer science curricula has often focused primarily on the distinct skill of programming due to the difficulty of teaching debugging to students [1]. Numerous studies reinforce the fact that past approaches have been limited by instructor time and a lack of personalization — in large course settings, instructors and even teaching assistants often cannot take time to interact with every student individually and guide them through developing a mental framework for debugging [1-2].

Now, generative AI and LLMs present a never-before-seen opportunity for personalized debugging instruction and early studies show that they can be highly effective for both the teaching and retaining of debugging skills if employed carefully [4]. Paralleling their benefits and drawbacks when applied to many disciplines, this same technology can serve to either dramatically improve skills or circumvent learning depending on how it is utilized [3].

Caution must be taken to ensure that students use AI in productive and beneficial ways so that they gain valuable skills rather than offload mental workload. When AI acts as a personal instructor rather than an answer key search engine, it can help students develop deep foundations for debugging. The ability to debug is a more important skill than ever before as LLMs continue to permeate codebases with an unprecedented quantity of code riddled with human-like mistakes [4].

It is clear that further studies and experiments analyzing the benefits of applying generative AI to debugging instruction and the provisions required to ensure these benefits are obtained are in order. Additionally, research on the retention of debugging skills after using AI chatbots should be conducted to ensure valuable independent skills, rather than tool reliance, is the outcome [2]. Even if chatbots are trained to respond exactly as a course professor or TA would, it must be proven that this unfettered access does not result in over reliance [3-4].

IV. References

[1] R. McCauley, P. J. T. de Raadt, R. T. Bergin, D. Thomas, J. B. Whalley, and M. L. Ratcliffe, “Debugging: a review of the literature from an educational perspective,” *Computer Science Education*, vol. 18, no. 2, pp. 67–92, Jun. 2008, doi: 10.1080/08993400802114581.

[2] S. Yang, M. Baird, E. O’Rourke, K. Brennan, and B. Schneider, “Decoding debugging instruction: A systematic literature review of debugging interventions,” *ACM Transactions on Computing Education*, vol. 24, no. 4, pp. 1–44, Sep. 2024, doi: 10.1145/3690652.

[3] S. Yang, H. Zhao, Y. Xu, K. Brennan, and B. Schneider, “Debugging with an AI tutor: investigating novice help-seeking behaviors and perceived learning,” in *Proc. ACM Conf. Int. Computing Educ. Res. (ICER ’24 Vol. 1)*, Melbourne, VIC, Australia, Aug. 2024, pp. 1–11, doi: 10.1145/3632620.3671092.

[4] Q. Ma, H. Shen, K. Koedinger, and S. T. Wu, “How to teach programming in the AI era? Using LLMs as a teachable agent for debugging,” in *Artificial Intelligence in Education (AIED 2024 Pt. 1)*, Recife, Brazil, July 2024, pp. 265–279, doi: 10.1007/978-3-031-64302-6_19.

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