A Video-Based Approach to Learning Debugging Techniques

Viktor Shynkarenko[1], Oleksandr Zhevaho[1]

[1] *Ukrainian State University of Science and Technologies, Lazaryana str. Dnipro, 49010, Ukraine*

Abstract

The presented research in line with the contemporary trend in education — microlearning, which involves using short videos to teach coding techniques and enhance the efficiency of the learning process. Microlearning is characterized by a student-centered approach, facilitates better knowledge retention, requires less time for learning, and allows for learning anytime and anywhere. Building on previous studies that developed a constructive-synthesizing model and corresponding software to track and analyze programmers' activities during coding and debugging in the Visual Studio environment, this paper introduces a developed system for visualizing debugging processes based on log file data to improve the effectiveness and efficiency of programming education. The tool reconstructs debugging sessions as videos with synchronized timestamps and explanatory comments, illustrating the sequence of actions during debugging and offering explanations and recommendations for improving the debugging process. The comments help students understand the logic behind specific debugging actions and provide tips on alternative approaches, fostering a deeper understanding of debugging strategies. During an experiment in the form of a debugging olympiad, log files contain all the information about the debugging processes was collected. The developed visualization system was tested based on this experimental data, confirming the accuracy of the tool in reconstructing sessions and generating appropriate comments. This concept of visualizing debugging processes has significant potential for improving the methods of teaching and learning debugging, offering substantial benefits for both instructors and students. Instructors can analyze individual and group debugging sessions to identify common errors and adjust their teaching methods accordingly. This approach helps instructors provide more personalized assistance, thereby improving students' debugging skills. For students, the ability to review their debugging sessions and receive contextual feedback helps to develop critical thinking and self-improvement skills.

Keywords

Software, debugging, visualization, constructive-synthesizing modeling, programming education, software engineering, video-based learning

1. Introduction

Systematically checking programs for errors, identifying, and fixing them is the primary skill of software developers. Professional programmers spend between 20% to 40% of their time on debugging [1]. This percentage is significantly higher for beginners as they make more mistakes while coding and therefore need to debug their code more frequently.

Debugging training is often not given enough time, sometimes not even included in programming courses, and as a result, students are not familiar enough with debugging methods, concepts, and strategies that could potentially improve their debugging skills.

For example, novice programmers usually debug problematic programs without any plan. They tend to make changes randomly to reduce the difference between the erroneous and target programs, sometimes forgetting to revert incorrect modifications and introducing new bugs [2].

It is difficult for beginners to acquire excellent debugging skills simply by learning to write programs. Therefore, clear educational instructions should be provided to novices for learning

14th International Scientific and Practical Conference from Programming UkrPROGP'2024, May 14-15, 2024, Kyiv, Ukraine

: shinkarenko_vi@ua.fm (V. Shynkarenko); o.o.zhevaho@ust.edu.ua (O. Zhevaho)

0000-0001-8738-7225 (V. Shynkarenko); 0000-0003-0019-8320 (O. Zhevaho)

effective debugging strategies. Debugging training should take place at an early stage so that novices do not attempt to debug their programs using improvised and non-systematic methods.

Although different debugging models may contain various subtasks, they can be grouped into four main categories: problem identification (comparing expected and actual program outputs), error localization, error correction, and solution verification. These four subtasks represent the fundamental steps developers usually face during debugging. Utilizing a systematic step-by-step approach to debugging will enhance debugging skills, increase debugging speed and accuracy, and reduce the number of introduced errors [2].

Many students require assistance with debugging to make progress when learning how to write programs. While personal interaction with students to give feedback on their programs is effective for their learning, it is becoming more challenging due to the increasing number of students and distance learning. Instructors cannot work synchronously with students to provide assistance when each student is learning at their own pace.

Online learning has significantly changed the way teaching is conducted, leading to a shift towards the use of new technologies. Videos have become one of the tools that teachers use to present educational content to their students. One way of using videos is to integrate them into lectures to visually illustrate concepts that may be hard to explain using only images and text.

There is a practical need to assist students with debugging, so that students' learning of programming wouldn't be complicated by their weakness in finding errors in programs.

We propose a new approach, a tool that will enhance teaching in introductory programming courses by providing students with debugging tips and teachers with a view to individual student learning trajectories.

The main idea of the approach is to create a comprehensive video visualization of the debugging process, using debugging process logs and supplementing them with visual hints. The comments serve as valuable annotations that help to understand the rationale behind specific actions taken during debugging and provide guidance on potential process improvements or alternative approaches.

2. Related work

2.1. Teaching Debugging

Despite the importance of debugging training, surprisingly little research focuses on debugging teaching strategies and developing debugging skills.

In a recent study, the relationship between programming experience and debugging behaviors in an introductory computer science course was investigated [3]. The findings revealed that students with prior programming experience tended to focus more on fixing style errors, while novice students concentrated on syntax errors. This research underscores the significance of tailored pedagogical approaches to support novice programmers in developing effective debugging skills.

Based on the necessity of tailored approaches, several studies conducted the teaching evaluation of a systematic debugging process [2, 4-7]. Many students are unfamiliar with the debugging process and strategies in general, and as a result, they use a trial-and-error approach. Teaching a systematic debugging process and other strategies during programming education can address this issue effectively. These studies developed a systematic debugging procedure based on the scientific method: observing the program's behavior, formulating hypotheses, testing them through experiments, and refining them as necessary until the cause is found. The research showed that the experimental group achieved significantly higher results in debugging tests, that demonstrate the usefulness and necessity of teaching debugging strategies.

Research emphasizes the importance of understanding students' debugging behaviors to tailor educational strategies effectively [8]. The findings suggest that educators should encourage

persistent trial-and-error efforts and provide timely interventions to support students facing difficulties. This approach aligns with the concept of productive failure, which reinterprets the challenges of debugging as valuable learning experiences [9].

Debugging develops a deep understanding and problem-solving skills in students. This perspective aligns well with educational theories emphasizing learning through iterative cycles of failure and correction [9]. The research highlights that debugging is not merely a hurdle but a crucial part of the learning process, leading to significant educational gains when approached correctly. It provides a foundation for developing teaching strategies that leverage debugging to foster computational thinking and resilience in problem-solving.

In addition to theoretical insights, practical investigations into debugging strategies further also illuminate this area. For example, submission log data from an introductory Computer Science course were analyzed to investigate the debugging strategies of novice programmers [10]. The research identified various debugging behaviors, including code deletions, comments, and print statements, and their relationship with relation to debugging efficiency. These insights into the debugging process can inform about the design and implementation of video-based learning environments aimed at enhancing students' debugging skills.

Furthermore, approaches aimed at supporting novice programmers in debugging tasks have emerged [11]. Recognizing the challenges faced by beginners in effectively debugging code, we developed tool that provides targeted guidance from the failure site to the fault location. This approach helps novices navigate the debugging process by reducing the problem space and offering informed guidance based on previous bug fixes.

Collectively, these studies provide empirical arguments in favor of teaching debugging strategies and techniques, and that the process of learning to program can be significantly facilitated by an effective debugging process.

3. Feedback and visualization

Timely feedback is essential for guiding students as they independently work on programming exercises. Since it is becoming increasingly unrealistic to expect one-on-one interaction between teachers and students, researchers have been developing various automated systems to facilitate learning process and provide prompt feedback.

In the past, extensive research has focused on providing automated assistance to students in learning to write programs, but there are relatively few works on providing automated debugging support for students. However, there are several tools that use a feedback-based debugging approach. One such tool is the Microbat, which runs in the Eclipse environment [12]. It utilizes a feedback-based debugging approach on specific trace steps to identify suspicious execution steps, effectively determining the root cause of errors.

A critical challenge in automatically generating feedback for programming tasks is developing hints that are as effective as those provided by teachers in person.

One of the program visualization systems that offers interactive personalized hints is TraceDiff [13]. TraceDiff compares the dynamic execution of incorrect and correct code, illustrating how an error leads to behavioral differences and where the trace of incorrect code deviates from the expected solution. Another visualization tool, VeDebug [14], combines video-based time-travel regression debugging with traditional debugging techniques.

An alternative approach to visualizing the software development process is using development environment extensions [15]. One such tool is Swarm Debugging that aligns with debugging visualization [16]. Integrated into the Eclipse IDE and leveraging the Java Platform Debugging Architecture, the system captures debugging events and employs visual tools like the Sequence Stack Diagram and Dynamic Method Call Graph to represent method invocation sequences and

their hierarchical relationships. These visualizations help developers quickly identify critical points in the code and understand the execution flow.

In addition to visualization systems, some researchers have created interactive development environments designed to teach students in introductory programming courses [17].

The use of video as a documentation option in combination with additional information has been used for GUI testing [18], where videos are recorded during test execution to supplement test reports.

One of the few systems providing practical automated debugging tips to support student learning is the Virtual Debugging Advisor (ViDA) [19]. Based on recognizing common errors in students' programs, ViDA has shown that a higher percentage of students were able to fix their errors with its assistance, and most respondents found ViDA useful for learning.

Despite the significant progress in developing automated systems for teaching programming and providing feedback, there remains a notable gap in research focused on automated debugging support for students. This indicates that existing work focuses mainly on teaching code writing rather than on the important step of software debugging. Therefore, increasing attention on creating and enhancing automated systems for program debugging is an important task for the further development of programming education.

4. Research aims and objectives

The purpose of this study is to develop tools for the automated creating of the debugging process video visualization using existing debugging logs, supplemented with visual cues. This will help to improve the process of teaching students how to debug.

To achieve this goal, the following tasks were set:

Develop a tool for visualizing the debugging process, clearly demonstrating the sequence of actions based on event logs received from the development environment.

Supplement the video with comments that provide explanations and suggestions for improving the debugging process.

Test the developed tool on the logs collected during the experiment.

5. Video visualization of the debugging process

In previous work, a constructor to generate debugging activity log files to formalize the data collection process regarding the usage of the integrated development environment (IDE) during debugging was developed using the constructive-synthesizing modeling (CSM) methodology [20]. This methodology can model and formalize various structures and constructive processes and has been frequently applied to software development and debugging processes [20-23].

An extension to the Microsoft Visual Studio was created based on the constructive model [20]. This extension records all debugging actions, along with their context, in the event log.

In this paper, we describe the advancements of these software tools. Using the developed models and tools, we have created a system to visualize the collected information, reproducing the sequence of actions during the debugging process.

To visualize the debugging process, we developed software consisting of the following components (Figure 1):

1. WebUI – provides a web interface for accessing system functions.
2. LogAPI – contains a RESTful API for receiving event logs from the IDE.
3. DataManagement – provides CRUD operations for working with log and video files.
4. LogDataProcessor – processes event logs received from the IDE. These logs serve as the basis for reconstructing debugging sessions and providing recommendations.

5. CommentGenerator – generates comments that explain events from the log file and recommend debugging actions. These comments are based on analyzing the debugging process and identify effective strategies and approaches, increasing the educational value of the video.
6. VideoGenerator – uses the processed log data and comments to generate a video representation of the debugging process. This video captures the user's interaction with the code, provides explanations and recommendations for debugging actions, and synchronizes with the timestamps from the log data. Periods of inactivity are skipped to make the video shorter and more informative. These moments are reflected in the comments.

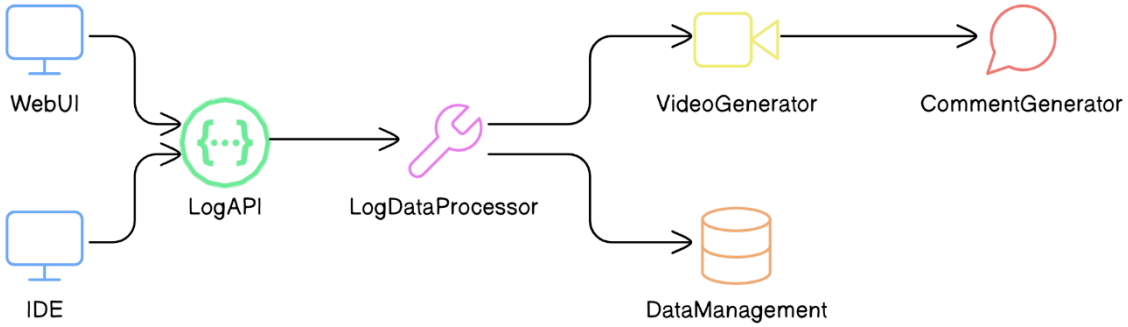

Figure 1: Tool components

The developed software tool aims to automate the creation of the software debugging process video visualization. It works based on event logs received from the IDE, which are generated in real time as the developer works with the program text. Each action is recorded with a timestamp and context, providing a chronological record of the debugging process. To record all events along with contextual information in the IDE, the tool presented in our previous work [20] is used.

By utilizing existing logs and supplementing them with visual cues, our approach offers a holistic view of the debugging process, clarifying the sequence of actions and strategies that developers use.

The video presentation of debugging activities is a valuable resource for both teachers and students. It allows students to observe effective debugging strategies in action, helping them develop fundamental debugging skills. The video is accompanied by tips and tricks that provide additional explanations for each step in the debugging process.

The debugging visualization tool interface (Figure 2) is designed to provide users with a comprehensive view of the debugging process by integrating: general information about debugging sessions (GeneralInfoSection), code display (CodeSection), and comments with recommendations (CommentsSection).

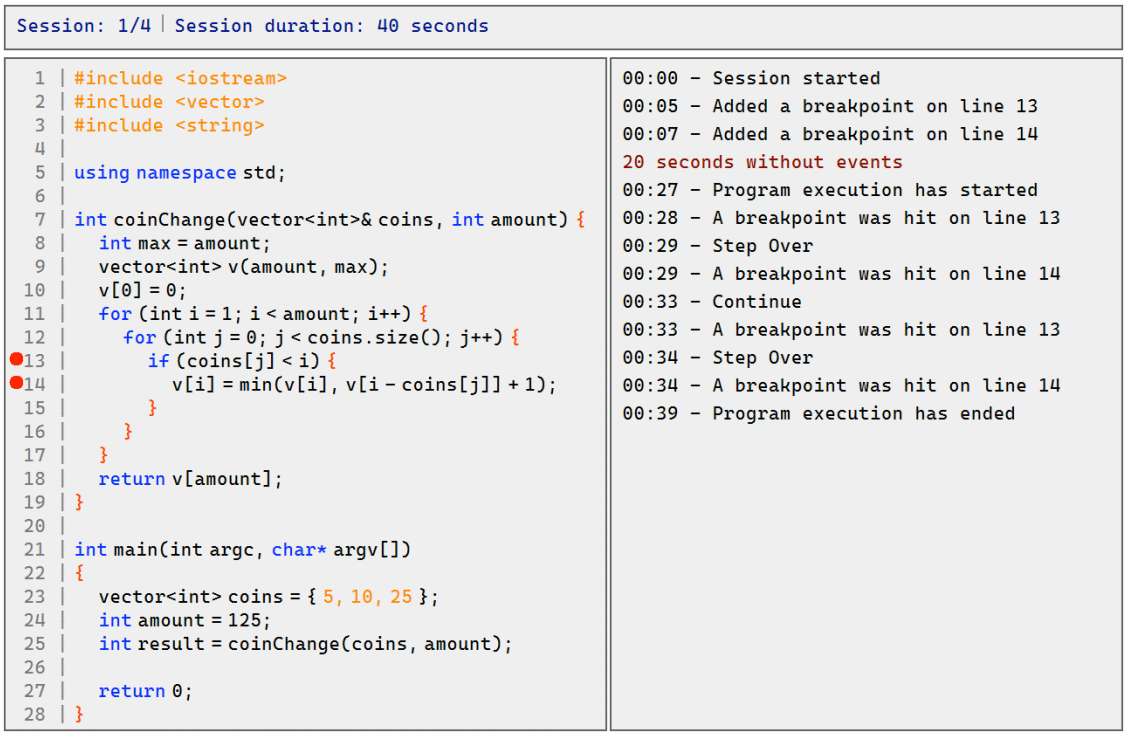

Figure 2: A snapshot of the debugging process visualization

The GeneralInfoSection includes the following details: the number of the current debugging session (in this example, the first), the total number of sessions (in this example, four), and the duration of the current session (in this example, 40 seconds).
The CodeSection occupies a prominent part of the interface, showing the source code being debugged and offering several features:

- syntax highlighting to improve readability;
- visual markers indicating breakpoints, with two modes: active (green) and inactive (red);
- highlighting of the currently executing line.

The CommentsSection, located next to the code display, contains comments and recommendations. It provides explanations for the actions performed during debugging and suggests alternative solutions. Each comment is timestamped with the log file timestamps and session duration, allowing you to visually track the debugging process. Recommendations and hints are highlighted in a different color to distinguish them from regular comments. During the debug visualization, corresponding comments appear dynamically to provide context and understanding of the process.

The debug visualization tool's interface is designed to be intuitive, informative, and user-friendly, offering a comprehensive and detailed view of the debugging process along with relevant comments and recommendations.

6. Validation

To verify the functionality of the developed tool, an experimental study on program debugging processes was conducted with students from various courses in the “Software Engineering” specialty at the Ukrainian State University of Science and Technologies. The experiment took the form of a debugging olympiad.

The process of common logical errors debugging by students was studied. A logical error occurs when the code is syntactically correct but produces incorrect results upon execution. These errors

are often complex and challenging for students to identify and resolve. To assess debugging skills, we employed the error seeding method. We developed thirty short programs in the C++ programming language, each containing one of the fifteen most common error types based on previous research [24, 25]. C++ was chosen because students have been acquiring programming and debugging skills in this language since their first year of study.

Each program contained a logical error confined to a single line of code, meaning the error could be fixed by modifying just one line. To aid in identifying errors, students were provided with sample input and output data.

Systematically exposing students to different types of errors can significantly enhance their debugging skills. Isolating individual errors simplifies the process of identifying and fixing problems.

The experiment involved 41 students from the first to the fifth year of study, all of whom had prior experience in C++ development and using the Microsoft Visual Studio IDE. The students were tasked with completing fifteen debugging exercises within a 4-hour period.

Participants are classified as “High”, “Middle”, or “Low” based on the number of correctly completed tasks (Figure 3).

Students in the “Low” class, if they have completed less than half of the tasks, there are 9 of them. The "Middle" class, comprising 23 students, completed more than half but not all the tasks. Nine students in the "High" class completed all tasks.

Thirty-two students, approximately 78%, managed to solve half or more of the tasks. Nine students corrected all errors, taking a little over two hours on average. No error category proved completely unsolvable for all students, and no category had more incorrect answers than correct ones. Regarding the difficulty of errors, the results show that most participants answered at least half of the tasks correctly. Most errors fell within the 75% correctness range. Students had moderate difficulties with misusing the sizeof function and misunderstanding variable scope, with 25% of tasks in these categories solved incorrectly.

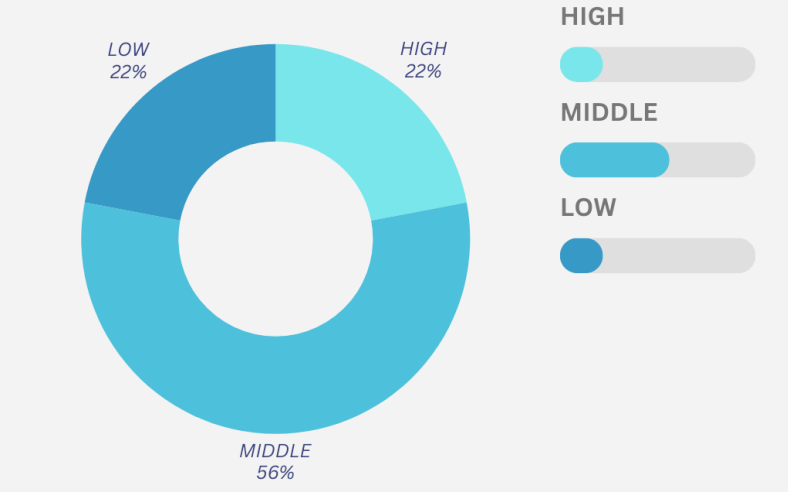

Figure 3: Participant classification

Figure 4 shows the list of the most frequently used commands of the Visual Studio development environment by the participants of the experiment.

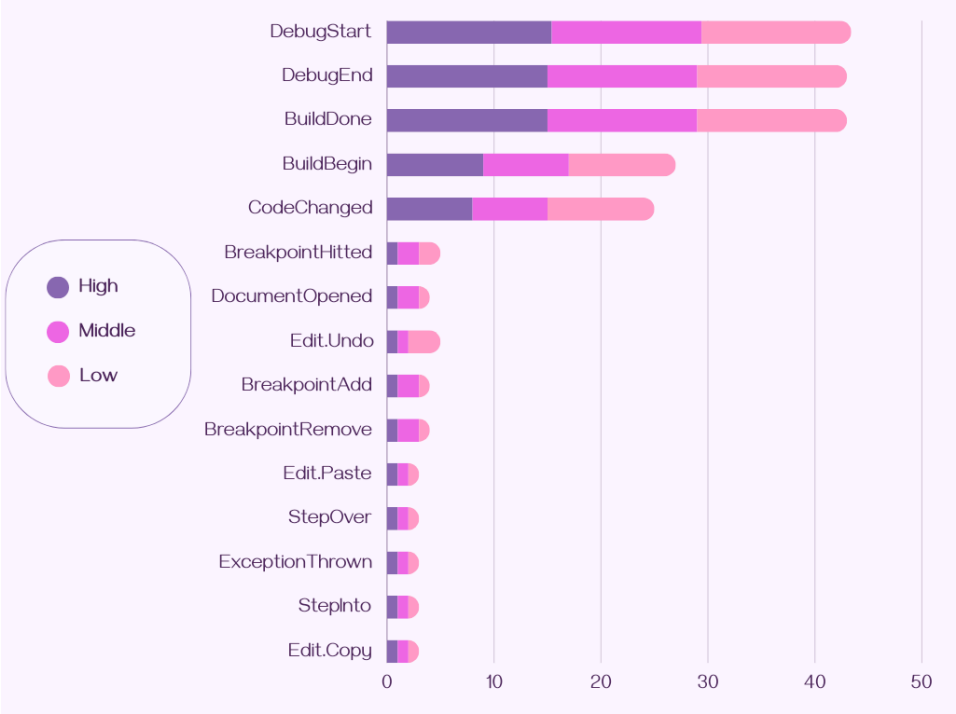

Figure 4: Frequency of using commands during the experiment

A small set of actions was common among almost all participants, with nine actions found in 90% of students. These actions included starting and stopping the debugger, changing code, executing the program, opening files with program text, and copying and pasting.

The results showed that “Low” class participants made significantly more code changes than others but had fewer debugging sessions. Additionally, "Low" class participants more frequently displayed variable values on the screen instead of setting breakpoints or using step-by-step execution.

Two types of breakpoints were observed: one to check if a part of the code is executed and another for step-by-step execution. Only "Low" class participants predominantly used breakpoints for step-by-step execution (86.67%), whereas "High" class participants used this method in only 25.71% of sessions. "High" class participants rarely used breakpoints and stepwise execution, suggesting their development skills are sufficient to form correct hypotheses without extensive debugging.

Figure 5 shows the statistics of debugging practices used by students during the experiment.

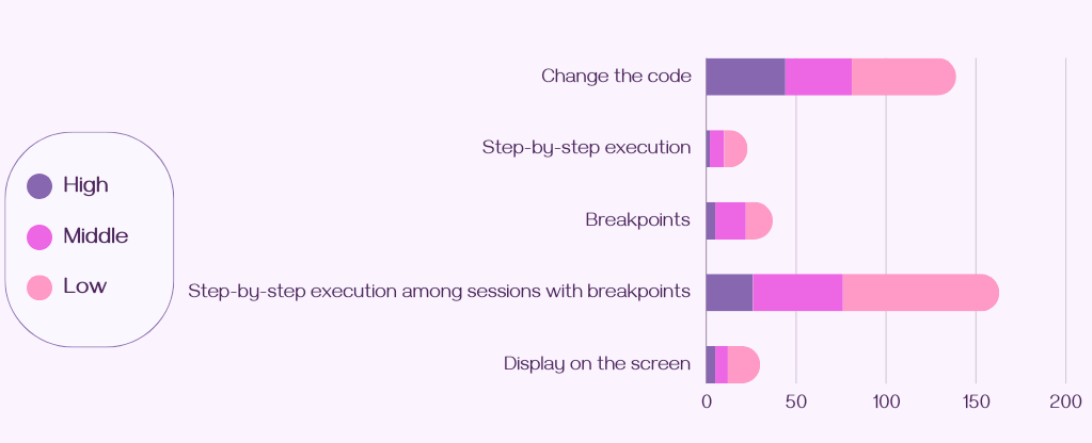

Figure 5: Statistics on the use of debugging techniques

Based on the most frequently repeated sessions, we identified four patterns of behavior among the experiment participants.

Pattern #1 – running the program in debug mode without making any changes or performing debugging actions. This pattern is used to check the program's behavior.

Pattern #2 – changing the program text and running it in debug mode without any further debugging actions. This pattern reflects the trial-and-error method, repeated until a solution is found or ideas run out. All participant groups use this method, though those in the “Low” category require more attempts and achieve poorer results.

Pattern #3 – running the program in debug mode with set breakpoints. This is used to study which parts of the program are executed and for step-by-step execution.

Pattern #4 – changing the program text, setting breakpoints, and running it in debug mode. This pattern corresponds to the error assumption method, where an error is assumed at a specific point, the code is modified, and breakpoints are set to verify the correction.

Most developers set breakpoints and traverse code using the debugger. Other debugging functions are used much less frequently by fewer developers. Only two participants utilized tools such as viewing variable values. Those who did not use debugging tools relied on trial and error and displayed values on the screen. These findings suggest a need for increased hands-on debugging experiences in curricula.

Although not surprising, the experiment demonstrated that experience is a key factor in debugging skills. All 4th and 5th-year students completed half or more of the tasks, whereas no first-year students completed all the tasks.

The data analysis also identified students who did not solve the tasks independently. One participant was disqualified for submitting pre-prepared solutions. Additionally, five more participants were not credited with correctly completed tasks due to suspicious processes lacking independent effort. Their solutions coincided with those of other participants, and their logs consisted of few debugging sessions with minimal actions, including large code pastes.

The experiment generated 487 event log files for 41 participants, totaling 2,415 debugging sessions and 16,536 events. Based on these logs, the functionality of the developed tool was tested. The verification phase assessed the accuracy of video display and commentary generation. The tool successfully reconstructed the debugging processes, accurately capturing the sequence of participants' actions. The resulting videos provided a clear visual representation of the debugging sessions. The test results confirmed the efficiency of the presented approach.

7. Conclusion

This paper presents a new approach that offers a practical solution for improving debugging skills through demonstrative debugging videos. The proposed method aims to enhance the efficiency and effectiveness of debugging training by combining visual representation with detailed descriptions of actions, comments, and recommendations. This approach enhances process understanding and facilitates knowledge transfer. Comments provide insights into the rationale behind certain actions, offering guidance on potential process improvements and alternative approaches.

Integrating video visualization with log file data offers a promising way to enhance the debugging learning process. By providing a comprehensive visual representation of the debugging process, novices can better grasp its intricacies, thereby addressing a gap in debugging training.

This tool is also beneficial for teachers, as it simplifies monitoring the development of students' debugging skills. By analyzing individual learning trajectories, instructors can assess each student's progress over time. Tracking improvements in debugging skills and identifying persistent problems allows instructors to adjust their teaching methods to better meet the needs of their students.

Furthermore, by studying patterns in debugging processes, teachers can identify common mistakes. This information can be used to develop teaching materials that address these problematic issues.

Future work could focus on expanding the tool's functionality and conducting additional research to further validate its effectiveness and usability.

References

[1] M. Perscheid, B. Siegmund, M. Taeumel, R. Hirschfeld, Studying the advancement in debugging practice of professional software developers, Software Quality Journal (2017) 83–110. doi:10.1007/s11219-015-9294-2.

[2] T. Michaeli, R. Romeike, 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, WiPSCE, ACM, Glasgow, 2019. doi:10.1145/3361721.3361724.

[3] J. D. Pinto, Q. Liu, L. Paquette, Y. Zhang, A. X. Fan, Investigating the Relationship Between Programming Experience and Debugging Behaviors in an Introductory Computer Science Course, in: Proceedings of the 5th International Conference on Quantitative Ethnography, ICQE, Melbourne, 2023, pp. 125–139. doi:10.1007/978-3-031-47014-1_9.

[4] X. Gao, K. F. Hew, A Flipped Systematic Debugging Approach to Enhance Elementary Students' Program Debugging Performance and Optimize Cognitive Load, Journal of Educational Computing Research (2023) 1064–1095. doi:10.1177/07356331221133560.

[5] C. Li, E. Chan, P. Denny, A. Luxton-Reilly, E. Tempero, Towards a framework for teaching debugging, in: Proceedings of the 21st Australasian Computing Education Conference, ACE, ACM, Sydney, 2019, pp. 79–86. doi:10.1145/3286960.3286970.

[6] A. Böttcher, V. Thurner, K. Schlierkamp, D. Zehetmeier, Debugging students' debugging process, in: Proceedings of the 46th Annual Frontiers in Education Conference, FIE, IEEE, Erie, 2016. doi:10.1109/FIE.2016.7757447.

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

[8] C. H. Liu, T. C. Hsu, Using a Hierarchical Clustering Algorithm to Explore the Relationship Between Students' Program Debugging and Learning Performance, in: Proceedings of the 2024 Joint of International Conference on Learning Analytics and Knowledge Workshops, LAK-WS 2024, CEUR-WS, Kyoto, 2024, pp. 13–22. URL: https://ceur-ws.org/Vol-3667/DC-LAK24-paper-2.pdf.

[9] Y. B. Kafai, D. DeLiema, D. A. Fields, G. Lewandowski, C. Lewis, Rethinking debugging as productive failure for CS education, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE, ACM, Minneapolis, 2019, pp. 169–170. doi:10.1145/3287324.3287333.

[10] Q. Liu, L. Paquette, Using submission log data to investigate novice programmers' employment of debugging strategies, in: Proceedings of the 13th International Conference on Learning Analytics and Knowledge: Towards Trustworthy Learning Analytics, LAK, ACM, Arlington, 2023, pp. 637–643. doi:10.1145/3576050.3576094.

[11] P. Ardimento, M. L. Bernardi, M. Cimitile, G. D. Ruvo, Reusing bugged source code to support novice programmers in debugging tasks, ACM Transactions on Computing Education (2019) 20(1) 2. doi:10.1145/3355616.

[12] Y. Lin, J. Sun, Y. Xue, Y. Liu, J. Dong, Feedback-based debugging, in: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering, ICSE, IEEE, Buenos Aires, 2017, pp. 393–403. doi:10.1109/ICSE.2017.43.

[13] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D'Antoni, B. Hartmann, TraceDiff: Debugging Unexpected Code Behavior Using Trace Divergences, in: Proceedings of the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, Raleigh, 2017, pp. 107–115. doi:10.1109/VLHCC.2017.8103457.

[14] B. Buhse, T. Wei, Z. Zang, A. Milicevic, M. Gligoric, VeDebug: Regression debugging tool for Java, in: Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: Companion, ICSE-Companion, IEEE, Montreal, 2019, pp. 15–18. doi:10.1109/ICSE-Companion.2019.00027.

[15] V. Shynkarenko, O. Zhevago, Visualization of program development process, in: Proceedings of the 14th International Conference on Computer Sciences and Information Technologies, CSIT, IEEE, Lviv, 2019, pp. 142–145. doi:10.1109/STC-CSIT.2019.8929774.

[16] F. Petrillo, Y. G. Guéhéneuc, M. Pimenta, C. D. S. Freitas, F. Khomh, Swarm debugging: The collective intelligence on interactive debugging, Journal of Systems and Software (2019) 152–174. doi:10.1016/j.jss.2019.04.028.

[17] J. Moons, C. De Backer, The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism, Computers and Education (2013) 368–384. doi:10.1016/j.compedu.2012.08.009.

[18] J. Shi, K. Schneider, Creation of Human-friendly Videos for Debugging Automated GUI-Tests, in: Proceedings of the 33rd IFIP WG 6.1 International Conference on Testing Software Systems, ICTSS 2021, 2022, pp. 141–147. doi:10.1007/978-3-031-04673-5_11.

[19] V. C. Lee, Y. T. Yu, C. M. Tang, T. L. Wong, C. K. Poon, ViDA: A virtual debugging advisor for supporting learning in computer programming courses, Journal of Computer Assisted Learning (2018) 243–258. doi:10.1111/jcal.12238.

[20] V. Shynkarenko, O. Zhevaho, Development of a toolkit for analyzing software debugging processes using the constructive approach, Eastern-European Journal of Enterprise Technologies (2020) 29–38. doi:10.15587/1729-4061.2020.215090.

[21] V. Shynkarenko, O. Zhevaho, Constructive modeling of the software development process for modern code review. in: Proceedings of the 15th IEEE International Scientific and Technical Conference on Computer Sciences and Information Technologies, CSIT, IEEE, Lviv-Zbarazh, pp. 392–395, 2020. doi:10.1109/CSIT49958.2020.9322002.

[22] V. Shynkarenko, O. Zhevaho, Application of constructive modeling and process mining approaches to the study of source code development in software engineering courses, Journal of Communications Software and Systems (2021) 342–349. doi:10.24138/JCOMSS-2021-0046.

[23] V. Skalozub, V. Ilman, V. Shynkarenko, Ontological Support Formation for Constructive-Synthesizing Modeling of Information Systems Development Processes, Eastern-European Journal of Enterprise Technologies (2018) 55–63. doi:10.15587/1729-4061.2018.143968.

[24] B. S. Alqadi, J. I. Maletic, An Empirical Study of Debugging Patterns Among Novices Programmers, in: Proceedings of the 48th ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE, ACM, Seattle, 2017, pp. 15–20. doi:10.1145/3017680.3017761.

[25] N. Alzahrani, F. Vahid, Common logic errors for programming learners: A three-decade literature survey, in: Proceedings of the 2021 ASEE Virtual Annual Conference, ASEE, 2021. doi:10.18260/1-2--36814.