Rethinking Debugging and Debuggers

Abdulaziz Alaboudi [*] **and Thomas D. LaToza** [†]

George Mason University, Fairfax, USA

Abstract

What is debugging? Despite dozens of studies of debugging, many important questions remain. Most existing studies are limited in focusing on studying debugging in the lab or through log data, leaving less known about how debugging occurs in everyday work. To fill this gap, we first identified a new source of data — live-streamed programming — with which to examine debugging in the field. Using this data, we then examined the activities developers do when debugging and the relationship between debugging and programming. This revealed many interesting findings. For example, navigation constitutes only 15% of debugging time, while editing and running code constitute a majority of time. These studies suggest that creating understanding of the issue and fix is central to the debugging process, a process shaped by how developers formulate and test hypotheses. We believe there is an important opportunity to create a new form of *hypothesis-based debugger*, which makes debugging hypotheses an explicit part of debugging tools which can be shared, queried against code, and drive views of code.

Keywords: Debugging. Automated debugging tools. Fault localization. Learning while debugging.

1 Introduction

For decades, debugging has been the subject of dozens of studies of developers. Researchers have investigated what makes debugging hard [1]–[3], the kinds of debugging related questions developers ask [4], [5], categories of strategies developers use to debug [6], [7], and specific tool features developers use to debug [8], [9]. These studies have revealed much about the nature of debugging, opening many new opportunities to improve the way we teach [10] and design tools for debugging. Modern debugging tools offer omniscient debugging [11], where developers can step forwards and backwards, slicers [12] for following control and data flow, and automatic fault localization [13] which uses the results of test execution to formulate guesses as to a defect's location.

However, our understanding of the nature of debugging is limited in important ways. Debugging is traditionally framed as primarily a problem of fault localization - the task of the developers is to find the line of code that they must modify in order to fix the defect. Tools as varied and diverse as the WhyLine [14] and spectrum-based fault localization [15] all have this as their primary goal. But little is known about just how big a role fault localization plays in the full activity of debugging. This is partly because debugging is often studied in isolation, separated from the broader activity of programming. And little is known about how the activities of debugging and programming are interrelated in the moment-to-moment work of developers. It remains unclear just how other key steps in the debugging process - such as formulating a hypothesis, proposing a fix, and understanding the implications of changes - fit into the broader activity of programming and debugging in everyday work.

A key barrier to improving our understanding of debugging is the lack of data on debugging activity. Most studies to date of debugging have been conducted either in the lab or through the analysis of log data from instrumented development environments. Lab studies enable imposing experimental control and carefully examining how dependent variables may be impacted by manipulated independent variables. However, lab studies are inherently artificial, reflecting an unfamiliar context in which developers work on unfamiliar code. While log data enables understanding debugging in naturalistic contexts, it is inherently limited in its ability to reveal developer intent. While log data can effectively answer questions about how frequently various tools are used, it is much more challenging to answer questions about developers' activities and intent when using each debugging tool.

To address these gaps, we have conducted a series of studies of debugging. From our observations of developers at work on open source projects [16], [17], we first created a new dataset of programming

[*]Email: aalaboud@gmu.edu
[†]Email: tlatoza@gmu.edu

PLATEAU
12th Annual Workshop at the Intersection of PL and HCI

DOI:
10.35699/1983-3652.2021.11

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

and debugging work, encompassing 3544 activities coded from 30 hours of development work [18]. We then used this dataset in several ways, examining the activities developers do when debugging and the relationship between programming and debugging. This revealed a number of interesting findings about the nature of debugging work. For example, navigation constitutes only 15% of debugging work, while editing (41%) and running (29%) constituted a majority of debugging time. Edit-run cycles are, even in today's development tools, surprisingly frequent: developers edit and run the program, on average, 7 times before fixing a defect and twice before introducing a defect [19]. And, in an experiment examining the role of hypotheses in debugging [20], we found that, while traditional automated fault localization tools offered no benefit in helping developers debug faster or more successfully, developers given information about potentially relevant debugging hypotheses were six times more likely to fix the defect successfully.

These studies suggest that debugging is much more than fault localization, with understanding of the issue and fix much more central to the debugging process. This suggests an important opportunity to rethink the nature of tool support for debugging work, thinking beyond just the step of fault localization to the process of understanding and reasoning about code changes that constitutes so much of debugging. In particular, given the centrality of finding a correct hypothesis that explains not only the location but the underlying cause of the defect, we believe there is an important new opportunity to create a new type of *hypothesis-based* debugger [21], which makes debugging hypotheses an explicit part of debugging tools and which can be shared, queried against code, and drive views of code.

2 Studying Debugging

Many studies have long investigated debugging behavior, beginning at least as early as 1974 [22]. However, the majority of existing studies were conducted in the lab [6], [23]–[25] or using log data of feature use in development tools [8], [26]–[29]. Lab studies test the impact of an intervention (e.g., a new debugging tool) or reveal developer behavior when faced with a specific task. While lab studies have an important and central role in science, they are inherently limited in the artificial contexts they create. Loggers instrument programming and debugging tools to record what developers do in real-world development work. However, log data has important limitations [30]. As it records only the use of specific features, without additional context it can be difficult or impossible to reconstruct developers' intent and determine, for example, if developer actions relate to a debugging or programming task.

In contrast, direct observational studies offer insight into how developers debug in naturalistic settings. Researchers observe debugging behavior with little or no interventions, revealing debugging as it naturally occurs everyday. Despite these advantages, naturalistic studies of debugging are surprisingly uncommon. Over the course of the past several decades, we identified only five direct observational studies of debugging [4], [5], [31]–[33]. This may be due, in large part, to the difficultly of gaining access to this data. Academic researchers require industrial partners to obtain this data, and confidentiality and privacy concerns can make this data hard to obtain and access. As a result, most of our understanding of debugging today remains based on log data of IDE use and lab studies.

To bridge this gap, we investigated a new method of studying development work: examining programming screencasts. Early use of screencasting was often intended to offer developers tutorial content, replacing traditional text-based documentation by explaining how to use development tools or new APIs [34], [35]. More recently, developers have begun to live-stream their own real-time work contributing to open source software projects [17]. These videos illustrate developers' work in action using their preferred development environment while working on real tasks in familiar and unfamiliar code. These videos are not rehearsed and aim to shows a direct view of the moment-to-moment behavior of developers engaged in real software development work.

To investigate the suitability of this data for use in studying debugging, we conducted a detailed qualitative analysis of live-streamed programming [17]. We found that developers work on open source projects in varying sizes, programming languages, practices, and domains. This offers opportunities to observe developers in diverse settings. Developers explain their work to developers watching, the sessions document both developers' actions and a running commentary, similar to think-aloud, describing why they are working as they do. Developers structure their sessions in a similar way. They first start the live-stream by stating a goal for the session. They then work towards this goal, reading documentation and writing, debugging, and running code. Most sessions depict work on open source projects, which developers link to in their session descriptions. Developers are occasionally interrupted, either by developers watching live or by others in their physical space, mirroring the typical interruptions developers face in their day-to-day work. After completing the session, developers may archive the video on platforms such as YouTube and Twitch, with most licensed under the Creative Commons.

3 Activities in Debugging

Using live-streamed programming videos, we created a new dataset of debugging and programming activity[1]. Using qualitative coding methods, we carefully designed a codebook capturing key programming and debugging activities, including navigating. For each activity, we also included details about this activity, such as how many files developers navigated and edited, the methods developers use to inspect program state, and the type of external resources developers consulted. After iterative refinement of the codebook to ensure it could be rigorously applied, we then applied it to 30 hours of development work. This yielded a large dataset of 3544 activities.

With this dataset, our first goal was to examine the totality of the debugging experience: from the moment developers first realize the behavior is incorrect (either through a reported issue or in their own programming) until they either fix the bug or decide to stop working. We call these *debugging episodes*. We found that the time developers spend in debugging episodes primarily involved editing (41%) and running programs (29%), together constituting over two-thirds of debugging time and four times as much time (15%) as navigating. Navigating is most common early in debugging episodes, along with running, while later activity involves more editing and consulting external resources. We observed that navigation was more common when in debugging episodes that started from a defect report rather than a defect inserted by developers while programming. Developers who debug a reported defect navigated between files six times more often. Despite these differences in the number of navigations between files, the navigation activity itself only constituted 19% of the debugging time for these episodes.

To follow up on these findings, we interviewed developers about their recent experiences with debugging. One theme that many debugging stories shared was learning while debugging. Developers struggled to understand why the code they have was not working as expected. Developers often described situations where debugging centered not around navigating code to localize a defect to a line of code but instead involved work to better understand their own codebase, third-party APIs, and users' needs. For example, one developer worked to debug a defect in a database endpoint. There was an error message, but it was not helpful. The developer started reading the source code and searched for a similar database endpoint that was not failing. The developer then learned how to correctly create an endpoint and connect it to the database by reading and tracing similar code. Another developer reported debugging a defect where he mistakenly believed that he was using the API correctly, leading to hours spent reading documentation.

The main takeaway from our investigation of activities in debugging is that debugging is more than just navigation. Developers did not spend the majority of debugging time navigating files of code. A central theme of our findings from interviewing and observing developers while debugging was that debugging is the process of learning and building mental models of why the program behaves incorrectly. This process involves editing code, running the program, reading documentation as well as code navigation.

4 Edit-Run Behavior in Debugging

In the previous study, we looked at debugging activities individually. We next examined how debugging activities together characterize edit-run behavior. Edit-run behavior occurs in both programming and debugging. When developers debug, developers formulate and test hypotheses about the cause of a

1 https://bit.ly/3qWdMVA

defect [4], [6], [31], [36]. To formulate a hypothesis, developers look for clues in the source code. To test a hypothesis, developers may edit the source code and run the program. Developers often test multiple incorrect hypotheses before fixing the defect [6], [20], which may result in multiple edit-run cycles.

To support edit-run cycles, live programming environments [37] aim to merge the edit and run steps into a single step [38], allowing developers to edit and run the program concurrently [39]–[44]. Lab studies of live programming tools have found that these tools can help developers comprehend and debug code more effectively [45], [46].

The shared vision of these tools is to provide a fluid development experience in which developers edit and observe the output with no interruptions. Most of the past work has eliminated the manual work of running the program by automatically generating program outputs for each program edit. However, is this enough for offering a fluid development experience for developers? We know little about how developers engage in edit-run cycles in their typical programming and debugging work settings to answer such a question. Investigating what other needs developers may need beyond automatically generating program outputs may reveal insight for future live programming tools that will better support edit-run cycles in debugging and programming.

Using the same dataset of live-streamed development work, we analyzed developers' edit-run behavior in both programming and debugging [19]. We mapped developers' activities to either an edit or run step, constructing 581 debugging and 207 programming cycles. We mapped the activity of editing and navigating files of code to the edit step. We then mapped the activity of running and inspecting the program to the run step.

We found that edit-run cycles in debugging lasted one minute, on average, and occurred seven times every 10 minutes. Cycles in programming were longer and less frequent, lasting on average three minutes and occurring twice every eight minutes. Developers edited only one file of code per cycle in 70% of debugging and 60% of programming cycles. These findings suggest that developers' edit-run cycle behavior is largely the same in programming and debugging. Developers themselves confirmed this, describing their editing and navigating behavior in programming and debugging as "similar".

Our analysis also revealed gaps where developers were interrupted from their work with other activities. These occurred both within and between edit-run cycles. Edit-run cycles with gaps were four times longer than edit-run cycles with no gaps. We identified four causes of these gaps, including working with scattered code, unfamiliar third party APIs, disintegrated development environments, and waiting to compile. These findings suggest the importance for live programming environments to move beyond simply easing the process of running the program to more broadly remove distractions from developers' edit-run workflow.

5 The Role of Hypotheses in Debugging

As developers debug, a first step towards understanding the cause of the defect is to formulate and test a hypothesis. A debugging hypothesis is a verifiable speculation about the possible cause of an incorrect behavior [4], [31], [36]. For example, a developer who sees a search feature fail might ask, "Why did the search not return the correct answer?". She might then hypothesize a cause and gather evidence to test it. Hypotheses capture the understanding a developer has as they approach a debugging problem, either correctly guiding a developer to gather evidence to understand and address a problem or incorrectly guiding them to gather evidence which leads no closer to understanding the cause of a defect.

Would a debugging tool that aids developers in formulating a correct hypothesis help in finding and fixing the defect faster? To better understand the role of hypotheses in debugging tasks, we conducted a lab study [20]. To observe how developers formulate and test hypotheses, we organized the debugging tasks into three *stages*, where developers were progressively given access to more information about the defect (bug report, source code, ability to edit and run). And we then elicited hypotheses at each stage. To simulate a future debugging aid, we provided participants with a list of potential debugging hypotheses along with a description of a strategy to test each hypothesis. The hypotheses were designed to be concise and describe a potential defect cause in general terms

Table 1. The potential hypotheses given to participants.

Task	Correct Potential Hypotheses	Incorrect Potential Hypotheses
1	**Hypothesis**: You are using the wrong DOM API or not using getElementsByClassName correctly. **How To Test**: Check if you are using the correct DOM API. Also remember: getElementById returns an HTML element that has the same id, get ElementsByClassName returns a HTML Collection that you have to iterate over.	**Hypothesis**: You are not using the correct CSS property. **How To Test**: Check if you are using the correct CSS property to change the visibility of an element. Example: visibility: 'visible' or 'hidden', or display: 'none' or 'block'.
2	**Hypothesis**: You are not binding the scope of 'this' for the callback. **How To Test**: Check that you are binding the callback for the click event with 'this'. For example: callback.bind(this).	**Hypothesis**: You are not creating the initial state. **How To Test**: Check if you have created initial state in the constructor. For example: this.state = {}.
3	**Hypothesis**: You are not setting the new animation position correctly. **How To Test**:Check that the new position value is set correctly and in pixels. For example: 15px.	**Hypothesis**:You are not assigning a callback to the click event. **How To Test**:Check if you have a callback function to the click event. JQuery has this pattern: $('#htmlId').click(callback)

applicable across many programs. Table 1 lists the hypotheses provided.

We found that developers struggled to formulate correct hypotheses by themselves. Developers formulated a median of two hypotheses for each defect. Early hypotheses were usually wrong. However, offering developers a set of potentially correct hypotheses made developers six times more successful in fixing defects. In contrast, offering developers potential fault locations did not help developers formulate a correct hypothesis or debug more successfully.

6 Hypothesis-Based Debuggers

Our studies suggest the importance of thinking more broadly about how debuggers might support debugging. We found that the activity of a debugging episode extends substantially beyond navigating code to localize the defect. Developers had to edit and run their program multiple times and read online resources while debugging. The goal was to arrive at a hypothesis that correctly explains the root cause of the defect. When developers received a list of potential hypotheses to test as they debug, they were six times more successful in fixing the defect.

However, existing debugging tools largely focus on supporting navigation activities to localize the defect. Tools like the Whyline [14], and Tarantula [15] support the developer in navigating within the source code to identify and examine potential fault locations. We found that offering developers an aid with potential fault locations did not significantly help developers in formulating correct hypotheses or in the overall process of fixing the defect. Developers who received a fault location aid struggled as much as developers who did not receive any aid. This suggests a clear need to consider how to support developers more directly in formulating and testing debugging hypotheses.

We envision a new type of *hypothesis-based* debugger, which helps developers in the process of collecting defect context and testing hypotheses. This process can be modeled as a decision tree. Each node within the decision tree represents a hypothesis to be tested, either by the developer or by a tool. Information may be gathered from a static query against the source code or by examining a runtime value. After gather evidence to accept or reject a hypothesis, this may lead to follow-up hypotheses, further narrowing down potential causes. Leaf nodes in the tree may then offer a full explanation of a cause of the defect.

We first illustrate the concept of hypothesis-based debugging through an a hypothetical example.

We then discuss how such an approach might be realized as a programming tool.

6.1 An Example Of Using Hypothesis-based Debugger

Sultan is a web developer on a team building an international food recipes application. He is working on a new feature in which users can add their private recipes. The application is implemented with a React[2] frond-end with connects to a Node.js[3] back-end through REST APIs. After implementing the add recipe functionality, Sultan discovers that the recipes list does not correctly update when the user clicks the add recipe button. To investigate, he first reads the source code for the click button. However, he does not see anything incorrect in his implementation. He next checks the callback handling the click event and sees nothing wrong as well. "Why does clicking the button not add the recipe to the list?" Sultan asks. He is unsure what might cause this behavior, making it difficult to generate further hypotheses.

Sultan decides to use a hypothesis-based debugger to suggest further hypotheses. The debugger first asks Sultan to reproduce the defect. Sultan refreshes the page, adds a new recipe in the form, and clicks on the add button. Sultan then tells the debugger that he is done reproducing. The debugger collects the source code that executed as well as run time execution values. The debugger extracts program context, such as that Sultan is using React, there is a button click, and there is network activity. Using this information, the debugger then searches for decision trees relevant to the defect. For each tree it finds, it begins traversing each tree and executing the hypotheses in each node. For brevity, we discuss one decision tree.

The first node has a hypothesis that "The event handler was executed". To test this hypothesis, the debugger uses a dynamic analysis of the execution, finding that the click handler was not executed. Based on this outcome, the debugger next considers the follow-up hypothesis at the left node: "The click handler was passed as a function". To test this hypothesis, the debugger queries an AST representation of the program and finds that it was indeed passed as a function. Finding evidence in support of this hypothesis, the debugger moves to the right. The third hypothesis states that "There is a UI element that obscured the button". As automatically testing this hypothesis is complex, the debugger offers instruction for Sultan to test it manually by inspecting the program's output. Sultan inspects the HTML layers and finds a hidden element which occludes the button. Sultan reports the result and confirms the hypothesis. The debugger then moves right to the leaf node, containing a hypothesis that explains the root cause of the problem:

"**hypothesis**: A UI element is visually blocking the button from receiving the user's click. Either remove the UI element or use Z-index to move it behind the button."

6.2 Collecting Defect Context And Testing Debugging Hypotheses

A key first step is for a hypothesis-based debugger to collect *defect context*, such as source code, program runtime values, API calls, and description of the incorrect behavior. Using this defect context, the debugger may then identify potentially relevant decision trees and traverse them collaboratively with the developer. Decision trees may be identified and traversed as follows:

1. Extract defect context from the source code, execution trace, and defect report
2. Use cues from the defect context to find relevant decision trees.
3. For each relevant decision tree, test a hypothesis, either by directly inspecting defect context or offering a strategy for the developer to make a judgement. This yields a binary result: True or False.
4. If the result is False, take the left branch. If the result is True, take the right branch. If a node is a leaf, stop.
5. Repeat steps 3-5 until a leaf node is reached.
6. Return relevant decision trees with the results of the investigation.

Hypotheses may be one of two types: an interior node describing evidence to be gathered to test a hypothesis and a leaf node describing a potential cause. Note that, in cases where there are no

2 https://Reactjs.org
3 https://nodejs.org

more hypotheses matching the evidence provided, a decision tree may end in a branch that has no leaf.

6.3 Challenges In Building Hypothesis-based debugger

Realizing this vision brings important challenges:

- *How many debugging decision trees are there?* In principle, it seems that the number of decision trees may be nearly limitless. However, we believe that, particularly for specific domains and technologies, there may be a more limited number capturing frequent issues. To explore this question, we have begun to examine React defects, identifying common issues developers may face.
- *What defect context is required to test hypotheses?* Hypotheses may vary widely, requiring a variety of information to test. In some cases, collecting information may be trivial. In others, it may be difficult or impossible to do automatically. By combining both machine and human intelligence, developers can be spared from the work of gathering evidence that can be easily automated while focusing on considering evidence requiring human judgement.

7 Conclusion

By studying debugging in the field, our studies have revealed that debugging involves not only fault localization but also the process of understanding the cause of a defect and formulating and testing a fix. Reconceiving the goal of a debugger as helping developers formulate and test hypotheses, rather than simply localize defects, suggests important new opportunities for programming tools to better support the activity of debugging.

References

[1] M. Eisenstadt, “Tales of debugging from the front lines,” in *Empirical Studies of Programmers: Fifth Workshop*, 1993, pp. 86–112.

[2] L. Layman, M. Diep, M. Nagappan, and R. a. DeLine, “Debugging revisited, toward understanding the debugging needs of contemporary software developers,” in *Empirical Software Engineering and Measurement*, Oct. 2013.

[3] Z. Coker, D. G. Widder, C. Le Goues, C. Bogart, and J. Sunshine, “A qualitative study on framework debugging,” in *International Conference on Software Maintenance and Evolution*, 2019, pp. 568–579.

[4] A. J. Ko, R. DeLine, and G. Venolia, “Information needs in collocated software development teams,” in *The International Conference on Software Engineering*, 2007, pp. 344–353.

[5] T. D. LaToza and B. A. Myers, “Developers ask reachability questions,” in *The International Conference on Software Engineering*, 2010, pp. 185–194.

[6] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, “Where is the bug and how is it fixed? an experiment with practitioners,” in *The Joint Meeting on Foundations of Software Engineering*, 2017, pp. 117–128.

[7] I. R. Katz and J. R. Anderson, “Debugging: An analysis of bug-location strategies,” *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, 1987.

[8] G. C. Murphy, M. Kersten, and L. Findlater, “How are java software developers using the eclipse ide?” *IEEE Softw.*, vol. 23, pp. 76–83, Jul. 2006.

[9] F. Petrillo, Y.-G. Guéhéneuc, M. Pimenta, C. D. S. Freitas, and F. Khomh, “Swarm debugging: The collective intelligence on interactive debugging,” *Journal of Systems and Software*, vol. 153, pp. 152–174, 2019.

[10] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, “Debugging: A review of the literature from an educational perspective,” *Computer Science Education*, vol. 18, no. 2, pp. 67–92, 2008.

[11] G. Pothier and É. Tanter, “Back to the future: Omniscient debugging,” *IEEE Software*, vol. 26, no. 6, pp. 78–85, 2009.

[12] M. Weiser, “Program slicing,” *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[13] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[14] A. J. Ko and B. A. Myers, “Finding causes of program output with the java whyline,” in *Conference on Human Factors in Computing Systems*, 2009, pp. 1569–1578.

[15] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *International Conference on Software Engineering*, IEEE, 2002, pp. 467–477.

[16] A. Alaboudi and T. D. LaToza, “Supporting software engineering research and education by annotating public videos of developers programming,” in *International Workshop on Cooperative and Human Aspects of Software Engineering*, 2019, pp. 117–118.

[17] ——, “An exploratory study of live-streamed programming,” in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2019, pp. 5–13.

[18] ——, “An exploratory study of debugging episodes,” *arXiv preprint arXiv:2105.02162*, 2021.

[19] ——, “Edit-run behavior in programming and debugging,” in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2021.

[20] A. Alaboudi and T. D. LaToza, “Using hypotheses as a debugging aid,” in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2020, pp. 1–9.

[21] A. Alaboudi, “Helping developers find and share debugging hypotheses,” in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2020, pp. 1–2.

[22] J. D. Gould and P. Drongowski, “An exploratory study of computer program debugging,” *Human Factors*, vol. 16, no. 3, pp. 258–277, 1974.

[23] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, “How programmers debug, revisited: An information foraging theory perspective,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, Feb. 2013.

[24] S. Jiang, C. McMillan, and R. Santelices, “Do programmers do change impact analysis in debugging?” *Empirical Software Engineering*, vol. 22, pp. 631–669, 2017.

[25] S. Baltes, O. Moseler, F. Beck, and S. Diehl, “Navigate, understand, communicate: How developers locate performance bugs,” in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2015, pp. 1–10.

[26] A. Afzal and C. L. Goues, “A study on the use of ide features for debugging,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 114–117.

[27] K. Damevski, D. C. Shepherd, J. Schneider, and L. Pollock, “Mining sequences of developer interactions in visual studio for usage smells,” *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 359–371, 2017.

[28] S. Amann, S. Proksch, S. Nadi, and M. Mezini, “A study of visual studio usage in practice,” in *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 124–134.

[29] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, “On the dichotomy of debugging behavior among programmers,” in *International Conference on Software Engineering*, 2018, pp. 572–583.

[30] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, “Programmers are users too: Human-centered methods for improving programming tools,” *Computer*, vol. 49, no. 7, pp. 44–52, 2016.

[31] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, “Studying the advancement in debugging practice of professional software developers,” *Software Quality Journal*, vol. 25, no. 1, pp. 83–110, 2017.

[32] S. Chattopadhyay, N. Nelson, Y. R. Gonzalez, A. A. Leon, R. Pandita, and A. Sarma, “Latent patterns in activities: A field study of how developers manage context,” in *International Conference on Software Engineering*, 2019, pp. 373–383.

[33] A. M. Vans, A. von Mayrhauser, and G. Somlo, “Program understanding behavior during corrective maintenance of large-scale software,” *International Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 31–70, 1999.

[34] M. Ellmann, A. Oeser, D. Fucci, and W. Maalej, “Find, understand, and extend development screencasts on youtube,” in *The International Workshop on Software Analytics*, 2017, pp. 1–7.

[35] L. MacLeod, M.-A. Storey, and A. Bergen, "Code, camera, action: How software developers document and share program knowledge using youtube," in *The International Conference on Program Comprehension*, 2015, pp. 104–114.

[36] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005, p. 480, ISBN: 1558608664.

[37] S. L. Tanimoto, "A perspective on the evolution of live programming," in *Workshop on Live Programming*, IEEE, 2013, pp. 31–34.

[38] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, and T. Pape, "Exploratory and live, programming and coding," *The Art, Science, and Engineering of Programming*, 2018.

[39] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[40] S. L. Tanimoto, "Viva: A visual language for image processing," *Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, 1990.

[41] S. L. Tanimoto, "A perspective on the evolution of live programming," in *Workshop on Live Programming*, 2013, pp. 31–34.

[42] S. McDirmid, "Living it up with a live programming language," in *OOPSLA Onward!*, 2007.

[43] S. Lau, A. Sarkar, S. Srinivasa Ragavan, and T. Barik, "Tweakit: Supporting end-user programmers who transmogrify code," in *Conference on Human Factors in Computing Systems*, 2021.

[44] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, and N. Tillmann, "It's alive! continuous feedback in ui programming," in *Conference on Programming Language Design and Implementation*, 2013.

[45] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Conference on Human Factors in Computing Systems*, 2014, pp. 2481–2490.

[46] S. Oney, B. Myers, and J. Brandt, "Interstate: A language and environment for expressing interface behavior," in *Symposium on User interface software and technology*, 2014, pp. 263–272.