Understanding Debugging as Episodes: A Case Study on Performance Bugs in Configurable Software Systems

MAX WEBER, Leipzig University, Germany
ALINA MAILACH, ScaDS.AI Dresden/Leipzig, Leipzig University, Germany
SVEN APEL, Saarland Informatics Campus, Saarland University, Germany
JANET SIEGMUND, Chemnitz University of Technology, Germany
RAIMUND DACHSELT, ScaDS.AI Dresden/Leipzig, Dresden University of Technology, Germany
NORBERT SIEGMUND, ScaDS.AI Dresden/Leipzig, Leipzig University, Germany

Debugging performance bugs in configurable software systems is a complex and time-consuming task that requires not only fixing a bug, but also understanding its root cause. While there is a vast body of literature on debugging strategies, there is no consensus on general debugging. This makes it difficult to provide concrete guidance for developers, especially for configuration-dependent performance bugs.

The goal of our work is to alleviate this situation by providing an framework to describe debugging strategies in a more general, unifying way.

We conducted a user study with 12 professional developers who debugged a performance bug in a real-world configurable system. To observe developers in an unobstructive way, we provided an immersive virtual reality tool, SoftVR, giving them a large degree of freedom to choose the preferred debugging strategy.

The results show that the existing documentation of strategies is too coarse-grained and intermixed to identify successful approaches. In a subsequent qualitative analysis, we devised a coding framework to reason about debugging approaches. With this framework, we identified five goal-oriented episodes that developers employ, which they also confirmed in subsequent interviews.

Our work provides a unified description of debugging strategies, allowing researchers a common foundation to study debugging and practitioners and teachers guidance on successful debugging strategies.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Human Factors, Program Debugging, Immersive Environment

ACM Reference Format:
Max Weber, Alina Mailach, Sven Apel, Janet Siegmund, Raimund Dachselt, and Norbert Siegmund. 2025. Understanding Debugging as Episodes: A Case Study on Performance Bugs in Configurable Software Systems. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE064 (July 2025), 23 pages. https://doi.org/10.1145/3717523

Authors' Contact Information: Max Weber, max.weber@informatik.uni-leipzig.de, Leipzig University, Leipzig, Germany; Alina Mailach, alina.mailach@informatik.uni-leipzig.de, ScaDS.AI Dresden/Leipzig, Leipzig University, Leipzig, Germany; Sven Apel, apel@cs.uni-saarland.de, Saarland Informatics Campus, Saarland University, Saarbrücken, Germany; Janet Siegmund, janet.siegmund@informatik.tu-chemnitz.de, Chemnitz University of Technology, Chemnitz, Germany; Raimund Dachselt, raimund.dachselt@tu-dresden.de, ScaDS.AI Dresden/Leipzig, Dresden University of Technology, Dresden, Germany; Norbert Siegmund, norbert.siegmund@informatik.uni-leipzig.de, ScaDS.AI Dresden/Leipzig, Leipzig University, Leipzig, Germany.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTFSE064
https://doi.org/10.1145/3717523

1 Introduction

Debugging, the process of identifying and resolving coding errors in software systems [Hailpern and Santhanam 2002; Katz and Anderson 1987; McCauley et al. 2008], is a tedious and time consuming activity. It is not just about fixing code, it also includes understanding why a bug happened in the first place and finding ways to prevent it in the future. If software developers can identify the root cause of a bug, they can fix it and improve the overall stability, reliability, and performance of their software systems. Performance bugs are particularly notorious. They degrade software efficiency and increase energy consumption in practice [Li et al. 2016; Velez et al. 2022], and detecting and fixing performance bugs is challenging, especially when bugs remain hidden until triggered by a specific software configuration. Thus, the debugging process of performance bugs in cofigurable software systems differs from traditional bug-detection strategies. Understanding *how* developers reason and investigate the root cause of a performance bug is crucial for devising and refining debugging strategies and for developing tools to support the debugging process. As a consequence, it is the human factor that is the focus of our study: How do developers approach debugging performance issues and what steps do they take?

While the debugging process for configuration-dependent performance bugs differs from traditional debugging, for which extensive research exists (see Section 2.1), it is advisable to leverage existing knowledge on useful debugging strategies to support developers also when debugging configuration-dependent performance bugs. However, in the literature, we find contradictory definitions of debugging strategies, and certain strategies often apply only in specific study setups [Chintakovid et al. 2006; Grigoreanu et al. 2006; Katz and Anderson 1987; Prabhakararao et al. 2003]. Moreover, strategies that resemble other identified strategies make them hard to distinguish. Such an unconsolidated landscape of debugging strategies hinders concrete suggestions of specific strategies or even starting points to debug configuration-dependent performance bugs.

Our goal is thus twofold: First, we want to understand what debugging strategies developers use while debugging performance issues of configurable software systems. Second, we want to carve out a framework based on fine-grained developer actions that allows for reasoning about developers' debugging activities, thereby setting existing debugging strategies in relation and enabling the identification for new ones. For this purpose, we analyzed debugging activities by means of an in-depth user study with 12 professional software developers. We observed their behavior while they traced the root cause of a configuration-dependent performance bug in a real-world configurable software system, using an immersive visual computation tool that we developed, called *SoftVR*, thereby collecting over 18 hours of video and audio material. *SoftVR* enables holistic observations of developers' activities, unlike traditional IDEs. It allows developers to position unlimited code windows freely around them and directly visualize control flow between the windows. Furthermore, developers can 'walk' through their code window setup, using the immersive capabilities of *SoftVR*. This way the code sections developers focus on are directly observable. The VR setup allows mental activities to be slowed down, such that they are easier to observe from an outside perspective [Andrews et al. 2010; Bandyopadhyay 2020; Lisle et al. 2021]. The study, including a pilot study with 10 students, spanned over 60 hours in total. We manually analyzed the resulting video and audio material using a fine-grained qualitative coding framework derived from existing debugging literature, followed by open coding to identify distinct goal-oriented debugging episodes.

As for the first goal, understanding debugging strategies for performance issues, we found that developers usually do not use a single debugging strategy throughout their debugging process, but a mixture of different strategies, often multiple strategies at a time, and sometimes we cannot even identify which strategy is being used. This situation hinders analyzing and comparing different

debugging approaches of the developers. As for the second goal, defining a framework that allows for reasoning about debugging strategies, we found *goal-oriented episodes* to be a more effective trade-off between a too broad classification into strategies and a too fine-grained one into individual actions. Describing developers' behavior in terms of what they aim to achieve in a time frame can clearly distinguish activities and strategies and allows for a better derivation of actions, such as tool support or education. This novel episode-based perspective provides a nuanced view on debugging, where not a single strategy is used and followed from the beginning to the end, but debugging is rather understood as a sequence of goal-oriented episodes that can make use of several strategic elements. This way, our framework allows for relating existing debugging strategies to each other, and therefore enables describing the debugging process of developers in realistic debugging scenarios. As a further notable contribution, we identified *aimlessness* as a phase of debugging that is prevalent, but not mentioned before in the literature. Developers need to overcome this phase for debugging configuration-dependent performance bugs successfully. This episode hindered developers in our study in detecting and understanding the bug. Not all developers were able to overcome this episode. We analyze the underlying reasons in Section 5 and propose novel recommendations to improve debugging tools, support education, and advance research on debugging approaches.

To summarize, in this paper, we make the following contributions:

- An overview of existing debugging strategies described in the literature.
- A fine-grained coding framework to track the actions developers perform during debugging. These actions are rooted in the debugging strategies proposed in the literature.
- *SoftVR*, a novel instrument for observing developers debugging real-world software systems in a highly controlled environment.
- A novel perspective and framework on the process of debugging, which supports better alignment of existing strategies and the definition of more nuanced approaches to debug.
- A replication package, including all data and code on a publicly accessible Web page[1].

2 Background and Related Work

In this section, we describe performance bugs as a notorious type of software bugs and provide an overview of debugging strategies described in the literature. We show that existing debugging strategies are not sufficient to find and reason about configuration-dependent performance bugs.

2.1 Performance Bugs

Performance bugs degrade user experience and increase energy consumption in practice [Li et al. 2016; Velez et al. 2022]. Numerous studies have shown that performance bugs are especially severe if they are configuration-dependent [Han and Yu 2016; Iqbal et al. 2022; Jin et al. 2012; Jovic et al. 2011; Nistor et al. 2013; Parnin and Orso 2011]. That is, only for a certain combination of (de-) selected configuration options, we may experience unexpected performance degradation. Such configuration-dependent performance bugs constitute up to 60 % of performance bugs [Han and Yu 2016; Han et al. 2018], with 80 % of these requiring code changes for resolution [Han and Yu 2016; Han et al. 2018]. For more than 60 % of configuration-dependent bugs, developers need, on average, five weeks to fix them [Krishna et al. 2020], making it a time-consuming process.

To illustrate the difficulty in finding and understanding a performance bug, consider the example in Figure 1: There is a bug in the image conversion tool Density Converter, which slows down the process of calculating a smaller image from a given image. The configuration option *fraction* is changed to a smaller value, which should result in a smaller image that is computed faster. However,

[1]Supplementary Web page: https://doi.org/10.5281/zenodo.14825306

computation took more time and the resulting image is larger. A profiler identified the method verticalFromWorkToDst as the computation hot-spot, which represents an extraordinary execution time at this specific code region. The bug is in the calculation of the baseWidth and baseHeight of method getDensityBucketsWithFractionScale, because dividing by a decimal number causes the new dimensions to increase instead of decrease. To pinpoint the root cause, a developer must traverse the control and data flow across different classes and methods. On the path from the hot-spot to the root-cause (i.e., the bug), there are eight functions within six classes that are not shown in the example but are the minimum number that have to be traversed by a developer. Therefore, developers have to open a large number of files in parallel and have to switch between methods within these files.

2.2 Debugging Strategies

To get an overview of the landscape of debugging strategies and to obtain a comprehensive overview of existing debugging strategies, we reviewed the literature using forward and backward snowballing from seven seed publications [Baecker et al. 1997; Eisenstadt 1997; Hailpern and Santhanam 2002; Hirsch and Hofer 2022; Velez et al. 2022; Xie and Yang 2003,?]. In total, we analyzed 74 publications and extracted 12 different debugging strategies. We focus on eight strategies that can be represented by our setup (shown in Table 1) and omit strategies that require additional artifacts (i.e., *offline analysis* [Böhme et al. 2017]) as well as strategies where multiple developers interact (i.e., *mod debugging* [Eisenstadt 1997]) or execute the program repeatedly (i.e., *system replication, testing* [Hirsch and Hofer 2021; Murphy et al. 2008]).

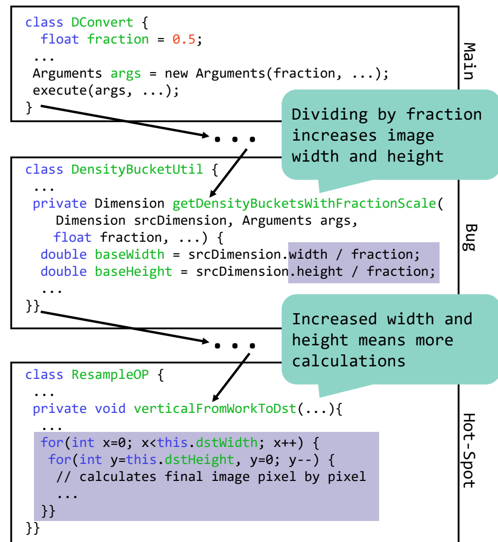

Fig. 1. Configuration-dependent Performance Bug in Density Converter, an image conversion tool.

For each selected strategy, we derived debugging actions expected to be observable during the user study, listed in column *Actions* in Table 1. These codes represent actions that developers perform when debugging. For instance, several studies report that developers use a scientific strategy, where they generate hypothese about the reason or fix for a bug and then test them, and we included the codes *state or test hypothesis*, and *accept or reject hypothesis*. This process led to 16 distinct codes derived from the literature. Actions are specific tasks developers perform when following a certain debugging strategy. Next, we describe the debugging strategies.

Program comprehension and *Inspect source code* are both recognized as essential debugging strategies in the literature. Program comprehension involves high-level browsing to build a comprehensive understanding of the program. This includes familiarizing oneself with the program's functions and structure by integrating insights from various representations, such as code, models, and log files, as investigated by Romero et al. [2007]. Velez et al. [2022] found that developers initially engage in high-level browsing searching for specific details, such as hot-spots and influential input variables, which can lead to significant time spent in configurable software systems. Vessey [1985] also observed this initial step of high-level browsing, showing that developers aim to understand the functionality and structure of a program before identifying and repairing errors. Johnson et al. [1982] described this as *system thinking*. Murphy et al. [2008] noted that program comprehension,

Table 1. Mapping from debugging strategies, VR behaviors, and experiment aspects to codes (actions) for video analysis. Debugging strategies and corresponding actions are extracted from the literature. For completeness, we added the actions for *VR* and *Experimentation* (explanation in Section 4.1).

	Title	Description	Actions	References
Debugging Strategies	Program comprehension	Developers gain comprehensive and high-level knowledge of the system by browsing through code and files, using available resources as needed.	hot-spot configuration bug report	[Murphy et al. 2008; Romero et al. 2007; Velez et al. 2022; Vessey 1985]
	Inspect source code	Examine the source code to understand its structure and functionality to build a comprehensive mental model of the code.	code reading scanning	[Böhme et al. 2017; Hirsch and Hofer 2021; Murphy et al. 2008; Peng et al. 2016; Subrahmaniyan et al. 2008]
	Planning	Developers create a structured plan or a series of tasks to be executed.	planning	[Subrahmaniyan et al. 2008]
	Simulation	Developers simulate code execution by mentally executing the code, thereby identifying potential issues through speculative reasoning.	simulate execution	[Eisenstadt 1997; Romero et al. 2007]
	Scientific strategy	Developers state and test hypothesis.	state/test hypothesis accept/reject hypothesis	[Eisenstadt 1997; Gould 1975; Perscheid et al. 2017; Romero et al. 2007]
	Intuition	Developers leverage past experiences and gut feelings to identify and resolve issues, based on instinctive insights.	follow intuition	[Böhme et al. 2017; Murphy et al. 2008]
	Follow data flow	Tracing objects and variables to understand how data moves and transforms within the system, thereby identifying potential issues.	follow data flow	[Murphy et al. 2008; Peng et al. 2016; Romero et al. 2007; Subrahmaniyan et al. 2008]
	Follow control flow	Tracking the program's execution path to understand the sequence of operations and identify deviations from expected behavior.	follow control flow	[Murphy et al. 2008; Romero et al. 2007]
VR	Spatial memory	Working with a VR interface reduces the strain on spatial memory and enhances cognitive abilities by allowing developers to interact with code in an immersive 3D space and improves navigating source code.	move player open window move window close window	[Cockburn and McKenzie 2002; Tavanti and Lind 2001]
Experimentation	Problem formulation	Participants are stuck and demand guidance from the study experimenter.	participants want help	
	Assistance	Participants have questions about the study setup, or they need to be reminded of the think-aloud protocol.	follow protocol intervention	
	Bug found	The experiment ends because the bug was found.	describe bug suggest bug-fix confirms bug	
	Bug not found	The experiment ends because participants are unable to find the bug.	experiment ends	

and the resulting mental models of the programs, are only build by successful novice developers, while less successful ones merely read the code without comprehending and gaining a deeper understanding. While *program comprehension* emphasizes a high-level understanding of the program, *inspect source code* focuses more on detailed examination of specific code segments. It is frequently mentioned in the literature as a strategy where developers read and analyze the source code. This debugging strategy is often a component of other debugging strategies [Böhme et al. 2017; Hirsch and Hofer 2021; Murphy et al. 2008; Peng et al. 2016; Subrahmaniyan et al. 2008].

Planning is a strategy where developers create structured plans, such as to-do lists, to systematically guide the debugging process. This can include meticulous inspection, such as found by

Subrahmaniyan et al. [2008], where developers track specific code sections and execution paths they had investigated and noted those still remaining, ensuring no part of the codebase was overlooked.

Simulation involves mentally simulating program execution to understand its behavior. This strategy requires significant time spent reading code and considering its dynamic aspects without directly executing it. Romero et al. [2007] found that participants focused on code windows, commenting on the program's dynamic aspects without actual execution. Similarly, Eisenstadt [1997] combines simulation, code inspection, and speculation in the *inspeculation* strategy.

Scientific strategy involves applying systematic, evidence-based methods to identify and resolve bugs. This strategy includes causal reasoning, hypothesis testing, and conducting controlled experiments. Several studies highlight how developers use scientific methods by setting breakpoints, examining program states, and formulating hypotheses. This way, developers can precisely identify and resolve issues by systematically exploring a program's behavior under controlled conditions [Eisenstadt 1997; Gould 1975; Perscheid et al. 2017; Romero et al. 2007].

Intuition draws on experience, pattern recognition, and sense of what "looks right" to developers to identify and resolve bugs. It leverages the developer's familiarity with common coding patterns and past experiences. It is used by novice developers as well as experienced programmers. While beginners often rely on intuition, particularly when they recognize patterns in the code that "don't look right" [Murphy et al. 2008], experienced developers often draw on their experience with similar problems and previous patches [Böhme et al. 2017]. Experienced developers use their intuitive sense of where issues are likely to occur, often bypassing more systematic methods in favor of quicker, experience-based solutions.

Follow data flow and *follow control flow* are widely used debugging strategies [Murphy et al. 2008; Peng et al. 2016; Romero et al. 2007; Subrahmaniyan et al. 2008]. Both involve tracing information through the program: Following the data flow focuses on tracking and reasoning about variable values, while following the control flow involves tracing the sequence of executed statements. Both strategies help novice and experienced developers identify discrepancies that may indicate bugs [Murphy et al. 2008; Romero et al. 2007].

Together, these studies provide a rich understanding of the strategies employed by developers during the debugging process. By examining both empirical studies and anecdotal evidence, this body of research sheds light on the varied and complex nature of debugging strategies, informing the development of more effective debugging strategies and tools. However, we found that existing studies are not consolidated in terms of identified debugging strategies, including different levels of abstraction for debugging strategies, contradictory definitions of debugging strategies that often apply only in specific study setups, or strategies that resemble other identified strategies, making them hard to distinguish. For instance, several studies report that developers use a scientific strategy [Eisenstadt 1997; Gould 1975; Perscheid et al. 2017; Romero et al. 2007], where they generate a hypothesis about the reason or fix for a bug and then test this hypothesis. However, this requires to comprehend the program, which is another identified strategy [Johnson et al. 1982; Murphy et al. 2008; Romero et al. 2007; Vessey 1985] and understand the code, developers need to reed code artifacts, which, in turn, overlaps with reported strategies of following data flow [Murphy et al. 2008; Peng et al. 2016; Romero et al. 2007; Subrahmaniyan et al. 2008] and control flow [Murphy et al. 2008; Romero et al. 2007]. Although these strategies have been independently identified and proposed, they clearly overlap. This overlap makes it difficult to determine which strategy is being used at any given time, complicating the development of tool support or training for specific strategies. In general, this lack of consolidation in the landscape of debugging strategies prevents us from recommending specific strategies or starting points for debugging configuration-dependent performance bugs. Thus, we develop a much-needed new view of the debugging process, described in detail in Section 4.

3 Study Design

3.1 Research Questions

Debugging is a complex, time-consuming, and often frustrating activity. It involves different cognitive demanding activities, including deciding whether a bug originates from a faulty implementation or is a consequence to be expected, for example, through activating or disabling functionality (e.g., by de-/selecting configuration options). Ultimately, developers need to understand the root cause of the problem. To support them, we equip developers with *SoftVR*, our immersive tool for debugging, and ask them to reason about a bug, during which we observe the different approaches that developers use while debugging. We pose the following research question:

RQ1 *What debugging strategies do developers use when debugging performance bugs?*

We found that existing debugging strategies in the literature are not consolidated. This complicates or hinders analyzing debugging strategies. Even if this is not the case, it might be beneficial to view the process of debugging on another level of abstraction, which is more consolidated and enables better derivation of actionables. Thus, we formulate a follow-up research question:

RQ2 *How can we describe the debugging process in a structured way?*

3.2 *SoftVR*: An Immersive Virtual Reality Tool

We use *SoftVR* as environment, as our study requires the ability to *directly observe how* developers debug. Traditional methods, such as think aloud protocols, can miss critical information, especially when investigating performance bugs, since IDEs do not provide all necessary information (i.e., hot spots, important configurations, control and data flow, etc.) in a single environment [Velez et al. 2022]. *SoftVR* leverages the advantages of virtual reality tools, offering immersiveness and unlimited 3D space, similar to other immersive systems [Andrews et al. 2010; Bandyopadhyay 2020; Lisle et al. 2021; Moreno-Lumbreras et al. 2024]. This allows developers to position code windows freely in the space and inspect them, making their interactions with the code and focus on specific areas observable, which is the main purpose of *SoftVR*.

SoftVR has two main components: (i) a call-graph visualization and (ii) specific code windows. First, the call graph provides an overview of the system's control flow, showing classes from the main method to the hot-spot; it can be positioned anywhere in the 3D space. Second, code windows can be opened from the call graph or by following method calls, with method calls highlighted in both the graph and code. This setup ensures that all necessary debugging information is available in one environment, so we can track developers' interactions and the information they pursue. Note that *SoftVR* is not meant for performance debugging in the wild, but as an experimental instrument for observing cognitive processes and developer behaviors. We provide all videos of the user study[2] illustrating the use of *SoftVR*. The video demonstrates how participants interact with the tool, highlighting its capabilities.

SoftVR has several advantages for experimenters and developers. First, there is unlimited space and an immersive nature of *SoftVR* (cf. Figure 1). Multiple code files can be displayed simultaneously without introducing cluttered, multiple hidden tabs. With the simultaneous display of files, we can directly observe developers' debugging strategies without having to trace their actions through opening, closing, and reopening code files.

Second, developers can exploit the unlimited space in *SoftVR* and structure multiple opened files according to their needs. This reduces the cognitive load, for example, by eliminating the need to decide whether to revisit closed files or by reducing the distance between two related files. Additionally, *SoftVR* can show multiple information in one place. In traditional debugging setups,

[2]The videos are available in our companion repository: https://doi.org/10.5281/zenodo.14825306

multiple views and tools are necessary to show code-level performance information, control and data flow information, and the call graph [Velez et al. 2022]. Switching between multiple tools puts unnecessary strain on the working memory [Andrews et al. 2010; Bandyopadhyay 2020; Lisle et al. 2021].

Despite its advantages, *SoftVR* presents several challenges. First, we must ensure that the developed tool and its features empower developers to debug efficiently. Given that most developers are used to traditional IDEs, *SoftVR*'s unfamiliar interface may threaten validity by requiring additional acclimatization time, potentially biasing results. Participants must therefore be given adequate time to familiarize themselves with the environment and training on how to use SoftVR's features. Additionally, an adequate number of breaks must be provided to minimize potential physical side effects of VR headsets, such as motion sickness or headaches. These factors increase the time and effort required for the study. To ensure that *SoftVR*, the training sessions, and breaks are mitigating inherent challenges of VR, we conducted two pilot studies (described in Section 3.4).

3.3 Study Setup

We have conducted our user study in a dedicated room at our institution. Participants put on the Valve Index VR headset with two controllers: one for navigating the VR environment and the other for interacting with the system. We provide a 3x3 meter obstacle-free space to move freely. All participants use the same setup to ensure consistent conditions.

In the *SoftVR* environment, participants begin in an empty virtual space with a ground floor and a distant horizon. They can access the call graph via a terminal attached to their left wrist. Code windows can be opened from the call graph or by following the control flow of existing code windows. Participants can physically move and turn around within the 3x3 meter area in real live and within the VR environment, as well as teleport in the VR space.

3.4 Pilot Study

To deal with the challenges introduced by *SoftVR* and its setup, we conducted two pilot studies. In total, nine PhD students and one professional designer tested our setup to ensure that tool, tasks, and study design are appropriate to answer our research questions. In the first study, we tested different components of *SoftVR*, ensuring that our implementation can be used for debugging purposes and identifying necessary changes to improve the tool. In the second pilot study, we focused on evaluating the study setup and procedure to verify whether the tasks, the time span, and the load on the participants is appropriate. In both studies, we used the same real-world software systems containing real-world bugs as used by Velez et al. [2022]: BerkeleyDB and Density Converter (described in detail in Section 3.7). In both studies, each participant debugged both software systems, one after another, with a bigger break in between. In total, we gained three insights that inform our final study design: First, after familiarizing themselves with *SoftVR*, several participants were able to successfully debug the software systems. In the main study, we therefore introduced a training task to ensure that developers were able to use *SoftVR* efficiently. Second, the bug in Density Converter is harder to find compared to the bug in BerkeleyDB, therefore, we use the first for the main data collection and the latter as a training task. Third, when participants had unlimited time to debug, several participants complained about physical side effects of *SoftVR*, such as headaches and motion sickness after 40 minutes. Additionally, participants who haven't found the bug after 35 minutes did not find it the time afterwards. In the main study, we therefore did not communicate a time limit, but ended the debugging process after 35 minutes to preserve physical and mental health that may occur using VR headsets.

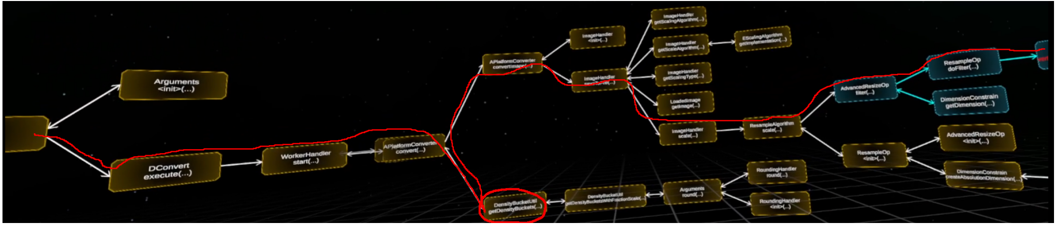

Fig. 2. Control flow graph of the subject systems Density Converter (excerpt). Red marks were added after the study to visualize the path from the main method to the hot-spot. Red circles show the bug position and red lines mark the path through the methods that has to be traversed to find the bug.

3.5 Participants

We recruited 12 software developers with professional skills, each with a minimum of three years of full-time development experience. Eight participants rated their career level as senior, four as junior. Except for two participants, all have experience with VR and rated their experience positive.

3.6 Procedure

Figure 3 depicts an overview of the study procedure. To avoid priming and bias, we told the software developers that they will debug in a virtual reality environment, not specifically mentioning debugging strategies. First, we welcomed participants and they completed a questionnaire on previous experience in debugging, their experience with VR systems, and some demographic information. After that, we introduced participants to the concept of performance bugs and to *SoftVR*. To keep the introduction similar for all participants, we used pre-recorded videos, ensuring as little variation between different participants as possible. Next, we fitted the VR headset, and participants familiarized themselves with the tool. We provided a tutorial in which participants followed instructions on how to interact with different components in *SoftVR*. Afterwards, participants completed a training task in which they had to find a bug in BerkeleyDB. We gave participants as much time as they needed to feel confident in handling *SoftVR*and understand the task.

Before the main task, participants were given a break to drink and rest. The instructions for the main tasks were exactly as the ones for the training task: participants received a bug report and were asked to search for the bug in Density Converter. Participants were additionally asked to follow the think-aloud protocol [Van Someren et al. 1994], and we recorded audio and video of the session. This way, we captured exactly what participants saw in VR alongside their think aloud audio stream. The main task ended either when the bug was found by the participant or after 35 minutes. Participants then filled out a questionnaire and we conducted a semi-structered interview. Finally, participants were debriefed, and we fully disclosed our research goals. We planned for a maximum study time of 2 hours (as shown in Figure 3) and the actual average study time was 90 minutes per participant.

3.7 Study Scenario

For both the training and main session we presented a realistic debugging scenario to help our participants understand the task and create an environment as realistic as possible. In this scenario, participants acted as a developer who received a bug report from a user of the corresponding software system. The bug report contains information under which configuration the user is running the system and what performance outcome seemed unusual to them. We then asked the

participants to use our tool to *decide and reason* whether the behavior the user observes, is in fact a bug, and where the bug occurs, or if it is an expected system behavior.

BerkeleyDB is a scalable high-performance database developed by Oracle. We use a bug report in which the user reports a loss in performance. In fact, the bug is due to the interplay of the two configuration options *Duplicates* and *Transactions*, which leads to locking of write transactions during parallel requests. Thus, the performance degradation is to be expected under the given configuration.

Density Converter is a multi platform image density converting tool developed by private open-source contributors. In contrast to BerkeleyDB, there is, in fact, an error in the code that leads to an erroneous lower performance. This bug depends on the configuration option *fraction*, which controls the size of the output image. This bug only emerges when users change the value of *fraction*, which complicates bug identification. Figure 2 shows the relevant part of the control flow. The hot-spot method contains two nested for-loops, one iterating over the target width and one over the target height of the picture. The bug is located in a sub-tree nine method calls away from the hot-spot. Here, the original height and width of the image is divided by the value of *fraction* and leads to higher target height and width of the image. Usually, an image should get smaller when provided with a smaller fraction. However, due to the division instead of a multiplication, the target dimensions becomes larger, meaning the for-loops in the hot-spot are executed more often and in turn, the system has higher energy consumption.

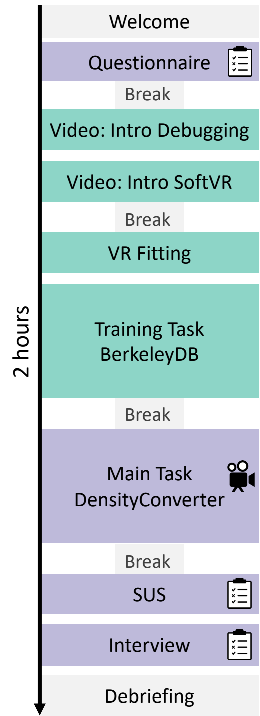

Fig. 3. Structure of the main user study. Purple boxes with icons indicate data collection points, green boxes represent introduction and training sections, and gray boxes show welcome, debriefing, and breaks.

3.8 Study Material

Additionally to demographics, audio, and video recordings, we collect the responses of the participants on the usability and conduct a semi-structured interview at the end of the study.

System usability scale. To gather insights into the usability of *SoftVR*, we use the *System Usability Scale* (SUS) as a traditional measure that delivers reliable results for small sample sizes, while at the same time being time efficient and concise [Bangor et al. 2008; Brooke 1996]. It is a ten-point Likert scale, five positive and five negative, for which our participants needed less than ten minutes to answer. For each participant, we calculate and report a single usability score across all items according to Brooke [1996].

Semi-structured interview. We developed an interview guide consisting of eight questions that target three different areas: The experience and usability of *SoftVR*, perception of debugging approaches in their usual day job and in *SoftVR* (RQ2), and subjective opinion and suggestions for improvement. The latter was included for future improvements of the tool, but is not analyzed for this paper. For each question, we document potential follow-up questions that the interviewer could use to get richer and more informative answers of participants. For analysis, we transcribed the interviews using whisper [Radford et al. 2023] generating an initial transcript that we corrected and refined during analysis.

4 Analysis Procedure

In this section we describe our two-step data analysis procedure. To analyze the audio and video data produced during the study, we used a two-stage approach: First, we analyzed all videos using a fine-grained coding framework that we derived from our literature review on debugging strategies, and we coded all material in 15 second video snippets. Second, we analyzed all the videos again to obtain a framework, in which we can relate existing debugging strategies and gain insights about the debugging process. We used open coding to identify goal-oriented debugging episodes.

4.1 Fine-Grained Coding

Our fine-grained coding framework is based on the literature in the area of debugging strategies, as described in Section 2. We generated codes for the coding framework by deriving individual actions that indicate the usage of a debugging strategy. These codes are displayed in Table 1, column "Actions", alongside the strategies from which they originate. Codes generally represent actions that developers perform when debugging. For instance, for the scientific strategy, we use the codes *state/test hypothesis* and *accept/reject hypothesis*. In total, this led to 12 distinct codes. Next, we added 11 further codes that describe actions performed in *SoftVR* (e.g., *move windows* or *move player*) and actions specific for user study scenarios (e.g., *participant wants help* or *suggest bug-fix*).

To analyze the videos, we split each recording into 15 second sections. Two authors coded the first three videos individually by deciding for each section whether one or more of the codes occurred. Then, we compared the codes: If we noticed a difference in the coding, the two authors watched the video sequence again, followed by a consensus-oriented discussion. After that, we coded two additional videos and calculated an inter-rater agreement between the two authors. We conservatively calculated the agreement by counting only codes that occurred at least once during the process: the two researchers agreed in 95.8 % of cases in the first, and 98.0 % of cases in the second video. This corresponds to Cohens's κ being 0.84 and 0.82 respectively, which usually indicates almost perfect agreement [Gisev et al. 2013; Landis and Koch 1977]. The videos of the remaining participants were then split between the two researchers and coded individually.

4.2 Coding of Debugging Episodes

Our literature indicated that the knowledge about debugging strategies is unconsolidated. That means that we can not provide guidance for developers when debugging performance bugs. To overcome this problem, we conducted an open coding without fixed time frames to identify specific episodes of debugging that are not captured within the fine-grained coding framework. We define a *goal-oriented episode* as follows:

> **Goal-Oriented Episodes:** An episode is the time interval in which the goal of a developer does not change. The transition between two episodes is always characterized by a change in the goal.

Naturally, whether a developer follows a specific goal is if not stated explicitly by the participants, a subjective interpretation of the researcher. The following factors aided in identifying debugging episodes and their corresponding goals:

- Participants were instructed to use the think-aloud protocol, which typically provides insights into their intentions. If they remained silent for a while, we prompted them by asking what they were doing to remind them.
- The viewing direction of the VR glasses and the laser pointer, attached to the right-hand controller, served as effective tools for tracking participants' focus.

- A sudden change in the participant's behavior—such as restless scanning, head tilting, or erratic movement of the laser pointer—often signals a search for a new focus. These behaviors vary per participant.

To minimize subjective bias and enhance consistency, two authors individually coded each video, marking the start and the end of each episode and documenting the goal and behavior of the developer. Then, the two authors compared the episodes and time frames. When the two authors were not agreeing on the goal of an episode, they had a consensus-oriented discussion. Within these discussions, the focus was on building a common understanding of the behavior of the developers. In some cases, this involved re-watching parts of the videos together, especially when the noted times differed by more than 10 seconds. After half of the videos, we started to revisit the found patterns and built a codebook consisting of descriptions of episodes that reoccure across different participants, which we then used for the remaining videos. Finally, we revisited the codebook again to ensure consistency across all participants and episodes.

5 Results

We start presenting the results with general observations of the behavior of our participants and the usability of *SoftVR*. Then, we dive into the research questions, presenting results and implications.

5.1 General Results

All developers could reach the bug location using *SoftVR* (see Table 2). However, five developers overlooked the bug or drew incorrect conclusions. Seven out of twelve developers successfully identified the bug and accurately reasoned about its root-cause. These results indicate that *SoftVR* and our study setup presented an appropriate level of difficulty for finding the performance bug.

This is also confirmed by the usability test that participants did at the end of the study. We measured usability using the standardized System Usability Scale (SUS) test [Bangor et al. 2008; Brooke 1996], with results shown in Table 2. The SUS grades range from A (superior performance) to F (failing performance), with C indicating average performance. Developers rated the usability of *SoftVR* as very good (indicated by 9 A- or B-grades). Additionally, three participants rated the usability as average.

Table 2. Outcome and usability judgements of Participants, including scores on the system usability scale (SUS), whether the bug position in the code was reached (Rea) and whether the bug was correctly identified (Des).

ID	SUS	Rea	Des
P1	B	Yes	Yes
P2	C	Yes	Yes
P3	C	Yes	No
P4	B	Yes	Yes
P5	A	Yes	Yes
P6	A	Yes	No
P7	B	Yes	Yes
P8	B	Yes	No
P9	B	Yes	Yes
P10	B	Yes	No
P11	C	Yes	Yes
P12	A	Yes	No

In post-study interviews, we gathered feedback from the participants on their experience with *SoftVR*. The majority of participants expressed positive sentiments during debugging, highlighting *SoftVR*'s functionality as a valuable aid in identifying and reasoning about performance bugs. Many participants also indicated that, with certain enhancements, such as the addition of text search functionality and the ability to pin code regions, they could envision incorporating a similar system into their daily workflows. Importantly, none of the suggested improvements were related to fundamental issues with VR itself. This indicates that the implementation was robust enough for a valid study, as participants focused on minor technical refinements rather than questioning the approach's feasibility.

5.2 RQ1: What debugging strategies do developers use when debugging performance bugs?

Figure 4 provides an overview of debugging strategies and actions used by the developers. On the left side, we depict the extracted strategies from our literature analysis (see Section 2.2). In

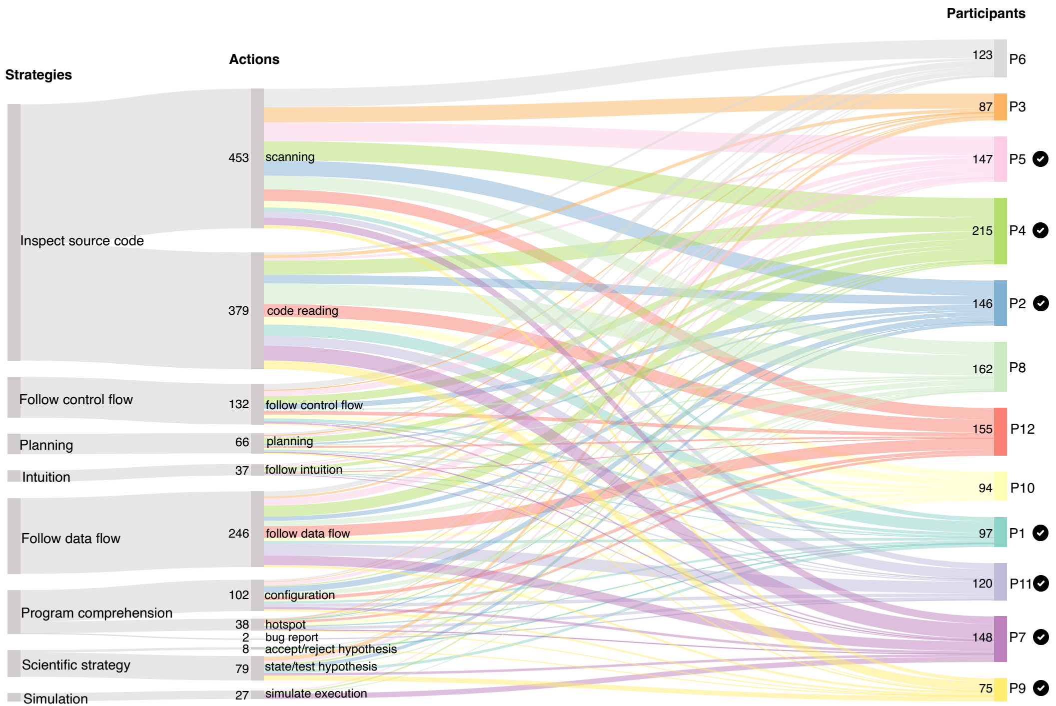

Fig. 4. Actions performed by the participants during debugging. The left column lists debugging strategies, the middle column shows the corresponding actions, and the right column lists participants (check marks indicate successful debugging). The bar heights and the numbers to the left represent the count of 15-second intervals in which developers performed a specific action.

the center, we state the derived actions per strategy that we used for coding the 15-seconds time frames. The numbers represent the number of times we found participants exercising this action. On the right, we show each participant with the number of 15-second time frames that have been recorded. The main result is the mapping from actions to developers and distribution of patterns. Both debugging actions *code reading* and *scanning* have been frequently used by all developers, since debugging is heavily based on reading and scanning code for information. The next most used actions are, *follow data flow* and *follow control flow*, which are also used by all developers. For these, participants have traced the bug through multiple files during the study. Interestingly, *simulate execution*, *planning*, and *follow intuition* were less commonly used during this study. Notably, 79 times, developers *state/test hypothesis*, but only 8 of them were answered (*state/test hypothesis*). Surprisingly, only P11 read the *bug report* to improve *program comprehension*.

A interesting observation is that most developers employ almost the full spectrum of debugging actions. That is, everyone used the following debugging actions: *scanning*, *code reading*, *configuration*, *hot spot*, *follow data flow*, and *follow control flow*. Only four participants (P4, P5, P7, P8) simulated parts of the program execution. Two participants (P11, P12) never formulated a hypothesis, three participants (P2, P3, P9) never followed their intuition, and three (P4, P5, P6) never formulated a plan. Overall, all developers used aspects of all debugging strategies from the literature.

5.3 Discussion of RQ1

Is it possible to infer which strategies developers pursue to find performance bugs, based on the extracted debugging actions? Considering Figure 4, we can clearly conclude that none strategy is recognizable. Instead, actions, coming from all strategies, are executed throughout the entire process and are intermixed for all participants. For example, *follow data flow* and *program comprehension* are both proposed as independent debugging strategies in literature, but our observations show that developers use them almost always in combination with *inspect source code* or *follow control flow*. Which strategy is actually performed and how they differ remains elusive. So, at least for performance bugs in a real-world configurable system, the proposed strategies are neither clearly recognizable, nor can we state which of them are successful. The same actions (and strategies) are performed in both, successful debugging processes and in failed ones. Based on this result, we cannot recommend any specific strategy to follow, nor can we clearly identify situations where further research or tool support might be beneficial.

Software configurability, as a key part of our study setup, requires developers to navigate and reason across multiple files and to follow data flows that span over distant and interdependent parts of the codebase. This characteristic makes debugging performance problems fundamentally different from addressing many functional bugs, where the root cause is often localized. Performance issues in configurable systems tend to emerge from the intricate interplay of diverse configuration options and system behaviors [Chi et al. 2003; Kolesnikov et al. 2019; Mühlbauer et al. 2023; Teng et al. 2006]. This context could explain why we observed frequent actions such as tracing both data and control flows and source code inspection during the debugging process. We argue that our study setup, specifically investigating configuration-dependent performance bugs, yield those problems which may also appear in other debugging scenarios.

Building on this observation, we question whether the proposed debugging strategies have an appropriate level of abstraction to describe or categorize approaches to debugging. The fluidity and intermixing of actions we observed across participants suggest that these strategies might oversimplify or overlook the nuanced, iterative nature of real-world debugging in highly configurable systems. This raises concerns about their practical applicability and prompts further inquiry into whether new frameworks or tools are needed to better support debugging in these complex contexts.

5.4 RQ2: How can we describe the debugging process in a structured way?

To answer RQ2, we employ open coding to identify the *intention* and the current *goal* of developers at different steps in the debugging process. By focusing on intentions and goals, we concentrate more on the human behavior (i.e., thinking) instead of the technical action (e.g., code reading). Thus, we lift the abstraction to a more goal-oriented, step-wise view of the debugging process. We identified and extracted different episodes (as described in Section 3). We show an overview of the debugging episodes in Figure 5 and a detailed view of the debugging episodes in Figure 6. In total, we identified five goal-oriented episodes that developers go through during debugging. We added two additional episodes, namely *knowledge seeking* and *pause*, which are caused by the experimental setting and not by the debugging-process. The most used episodes are *search*, followed by *exploration*, which together account for more than 80 % of the time. Note that the average duration of reasoning episodes is much longer than the average duration of search episodes. Next, we provide a description for each debugging episode:

Exploration. The *exploration* episode is an initial phase, in which developers seek to understand the scope and nature of the bug within the software system. This phase is characterized by gathering information, such as the hot-spot in the code or an influential configuration option, and gaining a

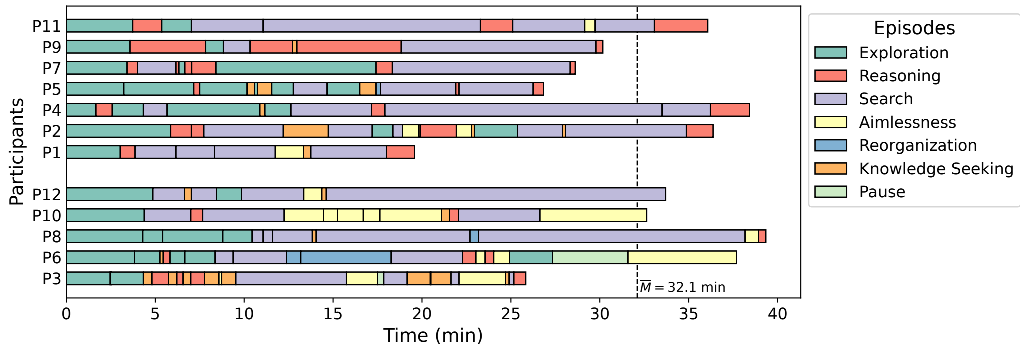

Fig. 6. Debugging episodes across the study timeline. The upper group represents participants who successfully found the bug, while the lower group did not. Different colors represent episodes, with consecutive episodes with the same color indicating a changed goal of the episode. The black dashed line ($\overline{M}$) shows the average time participants spend trying to find the bug in the main task.

comprehensive overview of the system to identify where issues may reside. The exploration episode is iterative and alternates with other episodes, often requiring several cycles of hypothesis, testing, and refinement to ultimately narrow down the root cause of the problem. The goal of this episode is to improve a developer's understanding of the issue's context, to obtain a list of potential causes, or to devise a plan for deeper, more targeted investigation in subsequent debugging episodes.

In our study, all developers start with the exploration episode to get familiar with the study specific information we provided (e.g., configuration options, hot-spot, etc.). Some developers (P1, P10) do not return to this phase later on in the debugging process. For example, P1 did an initial exploration and followed other episodes (i.e., reasoning, search, and aimlessness), without going back to exploring new information. Furthermore, exploration often transits into the search episode.

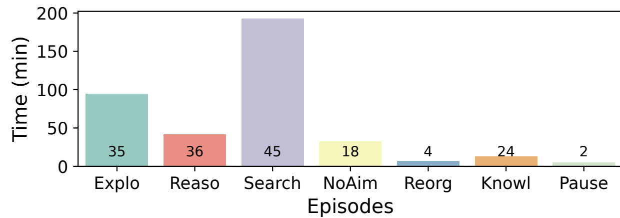

Fig. 5. Shares of individual debugging episodes on the overall debugging time for the episodes exploration (Explo), reasoning (Reaso), search (Search), aimlessness (NoAim), reorganization (Reorg), knowledge seeking (Knowl), and pause (Pause). The numbers in the bars denote the total number of times an episode occurs.

Search. In the *search* episode, developers actively seek answers to specific problems identified during earlier episodes of the debugging process. The search episode is, similar to exploration, iterative and requires persistence, attention to detail, and a methodical approach. That is, participants stated that they want to look for some specific information. For example, P6 stated after the exploration episode around Minute 8 *"I need to find the place where fraction is included."* (see P6 in Figure 6). By the end of this phase, the developer aims to have a (partial) understanding of the cause of the problem and an idea of what causes the bug. This phase is crucial for transitioning from problem identification to problem resolution in the debugging process, that is, all seven developers who successfully found the bug went from the search episode

to the reasoning episode, directly before they identified the bug. All participants used this episode frequently during the study to search for different information. This episode often follows directly after the *exploration* episode. After searching, developers transit into a *reasoning* or *aimless* episode.

Reasoning. Developers engage in detailed *reasoning* episodes to uncover the root cause of a bug. This can involve hypothesizing potential issues and making tentative guesses. The developer may examine the roles of various components and consider the high-level functionality of the software system and configuration options. They might simulate different scenarios to observe how the system behaves under various conditions. Throughout this process, the developer often questions internal mechanics and scrutinizes critical sections of the code, formulating hypotheses about the behavior of specific functions or algorithms. Discussions about observations and conclusions can occur, potentially leading to the identification of oversights in the code. For instance, P5 stopped at one point in their search (see P5 in Figure 6 around 22 minutes) and started reasoning about the system behavior, concluding that they might have gone too far and overlooked the bug. In the subsequent searching step, P5 stopped at the bug location and performed reasoned again, leading to the identification of the bug. Similar behavioral patterns occur for P4, P7, and P11.

Furthermore, despite uncovering certain insights, the developer might choose to set aside some complexities for later if immediate resolution seems impractical. For instance, P11 reasoned whether a specific variable (*numberOfThreads*) might be responsible for the bug. They did not take a final decision, but continued searching for the use the variable *fraction*, which, in the end, led to the correct identification of the bug.

Aimlessness. We found episodes in which developers could not find a line of reasoning to follow or have exhausted their initial ideas, both leading to a state of uncertainty. They have no clear direction or goal for identifying the bug. The process can involve aimless scrolling through the code base, reviewing unrelated sections of code, often, without a specific focus. This phase might include a sense of frustration and dwindling motivation as potential solutions seem increasingly elusive. Developers in this phase sometimes explicitly expressed their frustration such as P1 *"I am a bit lost trying to find the influence of fraction"* (see P1 in Figure 6 around minute 18). Developers may revisit previously read code areas, review configurations or the hot-spot code again, in the hope of stumbling upon a clue. This period is marked by the tension between a desire for a breakthrough and the increasing temptation to abandon the debugging process. If no new insights emerge, the developer eventually decides to step back from solving the bug. This was no unique case, as we counted 13 episodes of *aimlessness* with eight developers during our study.

Reorganization. Reorganization is specific to *SoftVR*, in that it allows developers to organize and interact with their source code in an immersive environment. This episode leverages the spatial capabilities of VR to enhance the debugging process, providing a unique and immersive way to manage code windows. We found that developers place code windows in different shapes and order to create a highly customized and efficient workspace that enhances the ability to identify, analyze, and orient within the code. Instead of reorganizing, we saw that P3, P4, and P10 spawned code windows directly at their desired positions, making it unnecessary to reorganize them, which can be seen by the absence of this episode in Figure 6.

Knowledge seeking. Developers communicate that they need some specific information to proceed with debugging. For example, they require assistance with interpreting Java syntax, particularly if they are more familiar with languages like Python or Go. In practice, developers may ask an LLM or just search the Web for such information.

Pause. Developers may be distracted from the debugging task due to the VR setup, such as when the VR headset causes discomfort or headaches after extended use. They lose focus on finding the bug, and when a participant had such problems during the study, we took a break, during which they took off the VR headset and waited until they felt better. In our study, two participants (P3 and P6) took a pause from the debugging task. Distraction episodes are not uncommon in experimental settings, as they may also occur in studies that involve additional hardware devices, such as fMRI [Peitek et al. 2021], brain wave [Campisi and La Rocca 2014], or eye-tracking studies [Peitek et al. 2018].

These seven goal-oriented episodes build a framework for understanding what developers intend to do during debugging a configuration-dependent performance bug. We identified three episodes that all developers have used while debugging, namely *Exploration*, *Search*, and *Reasoning*. Further, we found an episode of aimlessness, that slows down or hinders finding a bug.

5.5 Discussion of RQ2

The description of the debugging process in form of episodes, as illustrated in Figure 6, provides a structured overview and enables reasoning on different phases of debugging. We found that three episodes are part of every debugging session: *exploration*, *reasoning*, and *search*. Moreover, the final step of successfully identifying the root cause of the performance bug is always an episode of *reasoning*, where the prior episode comes from a *search* process that, in turn, was mostly triggered by a *reasoning* episode. Overall, *reasoning* appears substantially more often in successful debugging runs, while episodes of *aimlessness* seem to be a possible predictor for failures. However, selecting performance bugs may limit some episodes to performance-specific aspects that do not apply to functional bugs, such as aimlessness. That is, developers might be particularly prone to frustration and aimlessness due to the complexity of performance bugs.

Looking closer at the emerging success pattern of alternating *search* and *reasoning*, we found that the entire debugging process could be divided into two phases: (1) information gathering until clear hypotheses about the root cause can be made, and (2) a targeted search and reasoning to locate and verify the root cause of a bug. The first phase takes the most time, and it is here when participants fail. Thus, debugging support should concentrate on this phase, which includes *exploration* and *search*, to establish a hypothesis (i.e., not searching for the bug itself). If participants come to the hypothesis, it is likely that they also will have a successful *reasoning* [Alaboudi and LaToza 2020].

Also interesting are cases in which debugging failed. Here, most of the time, the *search* process ended in an episode of *aimlessness*. Entering this episode nearly always indicates failure. More than half of the participants (P1, P2, P3, P6, P8, P10, P11, P12) had an episode of *aimlessness*. For example, P1 was frustrated that they could not find the place in the code where the influence of the configuration option *fraction* applies to the dstWidth and dstHeight variable of the resulting image. While being on the right track, P1 followed the control flow only one step back and was then uncertain whether is would be beneficial to further traversal back on the call graph. Fortunately in this case, the reasoning was correct, but the search was too limited, as the bug was not one, but eight steps away in the call graph, and so the person continued the search. Based on participants' comments, we found that either the search took too long or participants could not find the information they were looking for (e.g., by expecting the root cause of the bug closer to the hot-spot code region). In addition, they may have had a prior wrong *reasoning* episode or an underspecified search target. These are interesting points for supporting the debugging process in the future in bringing developers back on the right track.

5.6 Interview Evaluation - Validating Goal-Oriented Episodes

At the end of the study, we conducted a semi-structured interview to obtain usability scores and general feedback on the study, as well as to let them reason on how they approach bugs in their daily life. Especially, we asked whether there is a difference in how they approached it using our setup to assess the validity of our study setup. We found that, when they talk about their own debugging experiences and approaches, they already use similar patterns in their descriptions to goal-oriented episodes, we identified. All developers describe their approach in terms of phases, in which they do different things, depending on what they want to accomplish. This is very similar to our definition of goal-oriented episodes, in which developers pursue a goal in an episode.

We found that participants mentioned all four episodes that are not specific to VR or the user study. That is, they described phases of exploration, search, reasoning, and aimlessness. Moreover, all participants describe a phase of problem-specific exploration at the beginning of their debugging process. For example, P4 stated: *"Normally, I try to get a fairly complete picture of something first."* and P9 asks themselves: *"Where does the error come from? What is the context?"*. Other developers start by exploring variable values and try to understand how they change, look at bug reports, exceptions, and stack traces or try to understand code in general, especially if it is not their own code. Most participants describe phases in which they search for specific information. They often describe search by following variables and investigating the changes that happen along the data flow. P9 stated that searching for information also depends on experience in debugging: *"At some point, you develop a feeling for it [where breakpoints need to be set]."*. Two participants genuinely addressed a phase of frustration and also suggested solutions on how to deal with it. P9 states that *"[...] there is no point in staying in such a frustrating moment."* and suggests to move forward fast when there is no obvious solution. P10 suggests to step back and revisit already available information: *"What I like to do is to start again from the beginning, perhaps with a clear head. In other words, abandon the current strategy and start again"*. Notably, P9 never entered a phase of aimlessness during the study. Some participants describe reasoning about potential problems as an important phase of debugging. P9, for example, does reasoning about values at breakpoints. P11 and P12 combine reasoning with formulating hypotheses what the problem could be. For example, P11 stated *"I need to understand what is going on in order to be able to formulate hypotheses"*.

The results of the interviews show that understanding debugging as goal-oriented episodes is close to how professional developers think about debugging. This encourages our proposal on formulating actions based upon this representation of the debugging process.

5.7 Actionables

Guided by the quantitative coding and qualitative interviews, we can derive several actionables for education, research, and tool support.

Education. Debugging is underrepresented in current CS curricula. Successful techniques on how to find bugs, how to reason on gathered information, and how to search through a code base or call graph are often missing. This is not surprising, as empirical evidence about successful measures is scarce. Even just showing these episodes of debugging and discussing with students the different phases and goals can be a valuable contribution to existing courses. Episode-specific tasks could strengthen the skills of exploration (i.e., system comprehension), search (i.e., systematic search for information in call and data-flow graphs, as well as in the code bases), and reasoning (i.e., formulating and testing hypothesis).

Research. It is time to consolidate existing research on debugging strategies. Our results neither disprove nor neglect the significance of existing strategies. Instead, we call for a more integrated

approach that reasons how strategies relate to each other, how they are applicable in which scenarios, and how to best identify them in practice. With our experiment and framework, we make a first step in this direction, and we call the community to start a joint community effort to systematically understand debugging.

Tool support. There is support for debugging in IDEs, such as call graphs, variable settings, and find usages. However and especially in the context of performance and configurability, which add further dimensions of complexity when locating bugs [Velez et al. 2022], developers need more support. Concretely, an automated detection of episodes of *aimlessness* can be a strong tool to interrupt a path to failure and waste of time. Supporting the formulation of hypotheses and reasoning (e.g., via LLMs) seem a promising future direction in this regard. Our study provides detailed insights on future needs for tool support for developers during debugging.

6 Threats to Validity

In this section, we outline the threats to the validity of our study and explain mitigation.

We used a novel tool for conducting the user study and observing developers detecting and reasoning about a configuration-dependent performance bug. Usually, developers use an IDE for debugging and, to our knowledge, we are the first who utilized a VR environment for debugging (apart from early work that used AR for debugging [Reipschläger et al. 2018]). Our tool might influence developers in how they debug as they are not used to it. However, we implemented several measures to mitigate such effects: (1) We carefully tested our setup and tool by conducting two pilot studies; (2) we added multiple stages of training within our setup, ensuring that participants are able to use *SoftVR*; and (3) we conducted interviews after the study to ensure participants did not struggle with the tool or environment. Our pilot studies confirmed the feasibility of the study setup and demonstrated that the chosen real-world bug can be found by developers unfamiliar with the software systems' code.

Using qualitative analysis may limit internal validity as researchers might introduce bias based on personal experience and attitude. However, using qualitative methods was essential for addressing our research question and obtaining rich insights from video data. To mitigate subjectivity, we followed established methods of qualitative studies by creating an initial codebook based on existing literature and conducting both closed and open coding. Two researchers of the author team collaboratively coded the data, achieving a high inter-rater agreement, indicating that the coding is consistent across data. Additionally, assisting participants during the experiment could impact internal validity. However, interruptions occurred only in two cases: (1) reminding participants to use the think-aloud protocol if they were silent or too quiet; and (2) prompting them to explain their reasoning when identifying a potential bug. These interventions were defined based on insights from pilot studies, ensuring consistent study conditions for all participants.

Focusing on debugging real-world performance bugs ensures high ecological validity. Incorporating novice programmers, multiple systems, and diverse bugs would either require an infeasible number of participants or compromise internal validity by introducing additional confounding factors. Expanding the number of subject systems would similarly extend study duration or necessitate a significantly larger participant pool, challenging feasibility. Instead, by analyzing debugging in a real-world system rather than artificial tasks, we capture authentic developer behavior, strengthening the study's relevance. However, a broader research program incorporating these aspects could extend our findings and provide additional insights in future work.

7 Conclusion

Our study introduces a novel framework for understanding debugging processes of configuration-dependent performance bugs. Using *SoftVR*, we observed professional developers and identified five key debugging episodes: exploration, search, reasoning, aimlessness, and reorganization. This episodic approach offers a more detailed understanding of debugging, showing that reasoning episodes are more frequent in successful runs, while aimlessness often precedes failure.

Our findings have practical implications for education, tool developers, and practitioners. As for education, curricula can be improved by teaching specific debugging episodes to engage students in successful debugging. For tool development, insights into aimlessness provide opportunities for creating intelligent debugging assistants, while the significant time spent on searching suggests that improving search techniques could be of great benefit to developers while debugging. Practitioners can use our framework to analyze and optimize debugging processes, reducing time and costs.

This research advocates consolidating existing debugging strategies and encourages more integrated approaches for studying and refining debugging techniques. Future work should validate these episodes across various bug types and development environments and develop tools that leverage this framework for real-time debugging assistance. By offering a structured view of the debugging process, our work advances software engineering practice, especially in the complex realm of configuration-dependent performance bugs.

Another direction for future work is the comparison of VR-based debugging tools with traditional IDE-based debugging. In this study, we used *SoftVR* to observe developers during debugging, allowing us to analyze its applicability and effectiveness. This foundation now enables a direct comparison between *SoftVR* and conventional debugging tools to assess their respective advantages and determine which debugging scenarios benefit more from VR-based tools.

8 Data Availability

A comprehensive replication package is available at our companion Web page[3]. This package includes: (1) *SoftVR* training videos, (2) the anonymized videos of the debugging sessions (without the audio recordings), (3) interview transcripts, (4) our fine-grained coding framework, (5) the coding framework of debugging episodes, and (6) data analysis and visualization scripts. We could not include the audio recordings of the video of the debugging sessions, as they cannot be anonymized to protect participant privacy. Nevertheless, this package enables other researchers to reproduce our analysis, validate findings, and potentially extend our work, contributing to the broader understanding of debugging processes for configuration-dependent performance bugs.

Acknowledgments

This work has been partly funded: by the German Research Foundation (DFG) as part of Germany’s Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence “Centre for Tactile Internet with Human-in-the-Loop” (CeTI) and EXC-2068 – 390729961 – Cluster "Physics of Life" of TU Dresden, for projects SI 2171/3-1, SI 2171/2-2, AP 206/11-1, AP 206/11-2, and DFG Grant 389792660 as part of TRR 248 – CPEC; by the German Federal Ministry of Education and Research (BMBF, SCADS22B) and the Saxon State Ministry for Science, Culture and Tourism (SMWK) for the "Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig" (ScaDS.AI); and by the European Union as part of the ERC Advanced Grant “Brains On Code” (101052182).

[3]Supplementary Web page: https://doi.org/10.5281/zenodo.14825306

References

Abdulaziz Alaboudi and Thomas D LaToza. 2020. Using Hypotheses as a Debugging Aid. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, IEEE, Dunedin, New Zealand. https://doi.org/10.1109/VL/HCC50065.2020.9127273

Christopher Andrews, Alex Endert, and Chris North. 2010. Space to Think: Large High-resolution Displays for Sensemaking. In *Proceedings of the SIGCHI conference on human factors in computing systems*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/1753326.1753336

Ron Baecker, Chris DiGiano, and Aaron Marcus. 1997. Software Visualization for Debugging. *Commun. ACM* (1997), 44–54. https://doi.org/10.1145/248448.248458

Payel Bandyopadhyay. 2020. *Immersive Space to Think: the Role of 3D Immersive Space in Sensemaking of Textual Data.* Ph. D. Dissertation. Virginia Tech.

Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An Empirical Evaluation of the System Usability Scale. *Intl. Journal of Human–Computer Interaction* (2008). https://doi.org/10.1080/10447310802205776

Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where Is the Bug and How Is It Fixed? An Experiment with Practitioners. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 117–128. https://doi.org/10.1145/3106237.3106255

John Brooke. 1996. SUS - A quick and dirty usability scale. *Usability evaluation in industry* 189, 3 (1996), 189–194.

Patrizio Campisi and Daria La Rocca. 2014. Brain Waves for Automatic Biometric-Based User Recognition. *IEEE transactions on information forensics and security* 9, 5 (2014), 782–800. https://doi.org/10.1109/TIFS.2014.2308640

Eric Chi, A Michael Salem, R Iris Bahar, and Richard Weiss. 2003. Combining Software and Hardware Monitoring for Improved Power and Performance Tuning. In *Seventh Workshop on Interaction Between Compilers and Computer Architectures*. IEEE. https://doi.org/10.1109/INTERA.2003.1192356

Thippaya Chintakovid, Susan Wiedenbeck, Margaret Burnett, and Valentina Grigoreanu. 2006. Pair Collaboration in End-User Debugging. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 3–10. https://doi.org/10.1109/VLHCC.2006.36

Andy Cockburn and Bruce McKenzie. 2002. Evaluating the Effectiveness of Spatial Memory in 2D and 3D Physical and Virtual Environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 203–210. https://doi.org/10.1145/503376.503413

Marc Eisenstadt. 1997. My Hairiest Bug War Stories. *Commun. ACM* 40, 4 (1997), 30–37. https://doi.org/10.1145/248448.248456

Natasa Gisev, J Simon Bell, and Timothy F Chen. 2013. Interrater Agreement and Interrater Reliability: Key Concepts, Approaches, and Applications. *Research in Social and Administrative Pharmacy* 9, 3 (2013), 330–338. https://doi.org/10.1016/j.sapharm.2012.04.004

John D Gould. 1975. Some Psychological Evidence on How People Debug Computer Programs. *International Journal of Man-Machine Studies* 7, 2 (1975), 151–182. https://doi.org/10.1016/S0020-7373(75)80005-8

Valentina Grigoreanu, Laura Beckwith, Xiaoli Fern, Sherry Yang, Chaitanya Komireddy, Vaishnavi Narayanan, Curtis Cook, and Margaret Burnett. 2006. Gender Differences in End-User Debugging, Revisited: What the Miners Found. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 19–26. https://doi.org/10.1109/VLHCC.2006.24

Brent Hailpern and Padmanabhan Santhanam. 2002. Software Debugging, Testing, and Verification. *IBM Systems Journal* (2002). https://doi.org/10.1147/sj.411.0004

Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10. https://doi.org/10.1145/2961111.2962602

Xue Han, Tingting Yu, and David Lo. 2018. Perflearner: Learning From Bug Reports to Understand and Generate Performance Test Frames. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 17–28. https://doi.org/10.1145/3238147.3238204

Thomas Hirsch and Birgit Hofer. 2021. What we can learn from how programmers debug their code. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. IEEE, 37–40. https://doi.org/10.1109/SER-IP52554.2021.00014

Thomas Hirsch and Birgit Hofer. 2022. A systematic literature review on benchmarks for evaluating debugging approaches. *Journal of Systems and Software* 192 (2022), 111423. https://doi.org/10.1016/j.jss.2022.111423

Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2022. Unicorn: Reasoning about Configurable System Performance through the Lens of Causality. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 199–217. https://doi.org/10.1145/3492321.3519575

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88. https://doi.org/10.1145/2345156.2254075

Paul E Johnson, Frank Hassebrock, Alicia S Duran, and James H Moller. 1982. Multimethod Study of Clinical Judgment. *Organizational behavior and human performance* 30, 2 (1982), 201–230. https://doi.org/10.1016/0030-5073(82)90218-5

Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch Me If You Can: Performance Bug Detection in the Wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. https://doi.org/10.1145/2048066.2048081

Irvin R Katz and John R Anderson. 1987. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399. https://doi.org/10.1207/s15327051hci0304_2

Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. 2019. On the Relation of Control-flow and Performance Feature Interactions: A Case Study. *Empirical Software Engineering* 24 (2019), 2410–2437. https://doi.org/10.1007/s10664-019-09705-w

Rahul Krishna, Md Shahriar Iqbal, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2020. Cadet: Debugging and Fixing Misconfigurations Using Counterfactual Reasoning. *arXiv preprint arXiv:2010.06061* (2020).

J Richard Landis and Gary G Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *biometrics* (1977). https://doi.org/10.2307/2529310

Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. 2016. Automated Energy Optimization of HTTP Requests for Mobile Applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 249–260. https://doi.org/10.1145/2884781.2884867

Lee Lisle, Kylie Davidson, Edward JK Gitre, Chris North, and Doug A Bowman. 2021. Sensemaking Strategies with Immersive Space to Think. In *2021 IEEE virtual reality and 3D user interfaces (VR)*. https://doi.org/10.1109/VR50410.2021.00077

Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A Review of the Literature From an Educational Perspective. *Computer Science Education* 18, 2 (2008), 67–92. https://doi.org/10.1080/08993400802114581

David Moreno-Lumbreras, Jesus M González Barahona, Gregorio Robles, et al. 2024. Diving into Software Evolution: Virtual Reality vs. On-Screen. *International Conference on Software Maintenance and Evolution (ICSME)* (2024). https://doi.org/10.5281/zenodo.13662379

Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2085–2097. https://doi.org/10.1109/ICSE48619.2023.00176

Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the Good, the Bad, and the Quirky–a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin* 40, 1 (2008), 163–167. https://doi.org/10.1145/1352322.1352191

Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems Via Similar Memory-Access Patterns. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 562–571. https://doi.org/10.1109/ICSE.2013.6606602

Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209. https://doi.org/10.1145/2001420.2001445

Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 524–536. https://doi.org/10.1109/ICSE43902.2021.00056

Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C Hofmeister, and André Brechmann. 2018. Simultaneous Measurement of Program Comprehension with fMRI and Eye Tracking: A Case Study. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*. 1–10. https://doi.org/10.1145/3239235.3240495

Fei Peng, Chunyu Li, Xiaohan Song, Wei Hu, and Guihuan Feng. 2016. An Eye Tracking Research on Debugging Strategies Towards Different Types of Bugs. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. IEEE, 130–134. https://doi.org/10.1109/COMPSAC.2016.57

Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the Advancement in Debugging Practice of Professional Software Developers. *Software Quality Journal* 25 (2017), 83–110. https://doi.org/10.1007/s11219-015-9294-2

Shrinu Prabhakararao, Curtis Cook, J Ruthruff, Eugene Creswick, Martin Main, Mike Durham, and M Burnett. 2003. Strategies and Behaviors of End-User Programmers with Interactive Fault Localization. In *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*. IEEE, 15–22. https://doi.org/10.1109/HCC.2003.1260197

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. Robust Speech Recognition via Large-Scale Weak Supervision. In *International conference on machine learning*. PMLR, 28492–28518.

Patrick Reipschläger, Burcu Kulahcioglu Ozkan, Aman Shankar Mathur, Stefan Gumhold, Rupak Majumdar, and Raimund Dachselt. 2018. DebugAR: Mixed Dimensional Displays for Immersive Debugging of Distributed Systems. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI EA '18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3170427.3188679

Pablo Romero, Benedict Du Boulay, Richard Cox, Rudi Lutz, and Sallyann Bryant. 2007. Debugging Strategies and Tactics in a Multi-Representation Software Environment. *International Journal of Human-Computer Studies* 65, 12 (2007), 992–1009. https://doi.org/10.1016/j.ijhcs.2007.07.005

Neeraja Subrahmaniyan, Laura Beckwith, Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Vaishnavi Narayanan, Karin Bucht, Russell Drummond, and Xiaoli Fern. 2008. Testing vs. Code Inspection vs. What Else? Male and Female End Users' Debugging Strategies. In *Proceedings of the SIGCHI Conference on human factors in computing systems*. 617–626. https://doi.org/10.1145/1357054.1357153

Monica Tavanti and Mats Lind. 2001. 2D vs 3D, Implications on Spatial Memory. In *IEEE Symposium on Information Visualization, 2001. INFOVIS 2001.* IEEE, 139–145. https://doi.org/10.1109/INFVIS.2001.963291

Xiaolin Teng, Hoang Pham, and Daniel R Jeske. 2006. Reliability Modeling of Hardware and Software Interactions, and Its Applications. *IEEE Transactions on Reliability* 55, 4 (2006), 571–577. https://doi.org/10.1109/TR.2006.884589

Maarten Van Someren, Yvonne F Barnard, and J Sandberg. 1994. The Think Aloud Method: A Practical Approach to Modelling Cognitive Processes. *London: AcademicPress* 11, 6 (1994).

Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2022. On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support. In *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery. https://doi.org/10.1145/3510003.3510043

Iris Vessey. 1985. Expertise in Debugging Computer Programs: A Process Analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494. https://doi.org/10.1016/S0020-7373(85)80054-7

Min Xie and Bo Yang. 2003. A Study of the Effect of Imperfect Debugging on Software Development Cost. *IEEE Transactions on Software Engineering* 29, 5 (2003), 471–473. https://doi.org/10.1109/TSE.2003.1199075

Received 2024-09-13; accepted 2025-01-14